Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Personalización de entorno de desarrollo y despliegue sobre contenedores

Autor: Carlos Rodríguez Hernández
Tutor: Isabel Román Martínez

Departamento de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2019

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de Telecomunicación

Personalización de entorno de desarrollo y despliegue sobre contenedores

Autor:
Carlos Rodríguez Hernández

Tutor:
Isabel Román Martínez

Dep. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2019

Trabajo Fin de Grado: Personalización de entorno de desarrollo y despliegue sobre contenedores

Autor: Carlos Rodríguez Hernández

Tutor: Isabel Román Martínez

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2019

El Secretario del Tribunal

A Ana, a mis padres y abuela

Agradecimientos

A mis seres más queridos y cercanos: Mis padres, mi abuela y mi tía. Especialmente a Ana, quién ha movido cielo y tierra para hacer este largo camino más llevadero. Sin el apoyo de todos ellos en los malos momentos (que alguno ha habido) y su alegría compartida en los buenos, este momento no hubiera sido posible. ¡Gracias!

A todo el personal docente de la Escuela, por sus enseñanzas durante las clases y prácticas; especialmente a Isabel Román por su paciencia, consejos y apoyo a lo largo del desarrollo de este Trabajo.

A todos los compañeros de clase que he conocido a lo largo de estos años, especialmente a aquellos que pasaron de compañeros a amigos. Sin ellos hubiera sido algo más difícil.

A los diferentes compañeros de trabajo, y ahora también amigos, con los que me he cruzado durante mi experiencia laboral. He aprendido mucho de todos y cada uno de ellos.

Carlos Rodríguez Hernández

Sevilla, 2019

Resumen

El objetivo de este proyecto es diseñar, desplegar y configurar un entorno de desarrollo y gestión de software para un grupo de trabajo colaborativo.

Este entorno abarca diferentes fases del ciclo de vida de un proyecto software, incluyendo el desarrollo del código, pruebas, QA[1], automatización, despliegue en producción, etc. El entorno se ha dotado de un sistema de gestión de tareas, planificación temporal, revisión de código y otra serie de características innatas en proyectos colaborativos llevados a cabo por equipos multidisciplinares en los que coexisten diversos roles más allá de los desarrolladores, como pueden ser; jefe de proyecto, diseñadores gráficos, ingenieros de calidad, documentadores técnicos, etc.

Por tanto, y a grandes rasgos, el proyecto consta de un entorno de desarrollo desplegado en contenedores Docker que proporciona las herramientas adecuadas a cada miembro del equipo de trabajo según sus objetivos y tareas. Esto es complementado con una aplicación centralizada, GitLab, a la que tienen acceso todos los miembros del grupo de trabajo y que sirve como nexo común para el trabajo grupal, alojando el código, así como el estado de los diferentes proyectos.

[1] Del inglés "Quality Assurance", se refiere a una serie de procesos que debe seguir el software para cumplir una serie de requisitos en términos de eficiencia, seguridad y concordancia con las necesidades del cliente o usuario de dicho software.

Abstract

This project aims to implement a software development and management environment for a collaborative workgroup.

This environment covers different phases of the life cycle of a software project, from code development, testing, QA, automation, production deployment, etc.; not only focused on the technical part, but it has also been equipped with a task management system, time planning, code review and other features of collaborative projects in which different members with different profiles participate.

Therefore, the project consists of a local development environment based on Docker containers where each member carries out the software development, totally or partially; this is complemented by a centralized application, GitLab, to which all the members of the team have access and which serves as a common link for the group, hosting the code, as well as the status of the different projects.

Índice

Agradecimientos **ix**

Resumen **xi**

Abstract **xiii**

Índice **xiv**

Índice de Ilustraciones **xvi**

Índice de Bloques de Código **xix**

Glosario de términos **xxi**

1 Introducción **23**
1.1 Motivación y contexto *23*
1.2 Objetivos *23*
1.3 Escenario propuesto *24*
1.4 Ventajas del escenario *24*
1.4.1 Desde el punto de vista del desarrollador 25
1.4.2 Desde el punto de vista del proyecto 25
1.5 Desventajas del escenario *26*
1.6 Estructura de la memoria *27*

2 Estado de la técnica **28**
2.1 Gestión del ciclo de vida del software *28*
2.1.1 Comparativa de herramientas 29
2.2 Control de versiones *30*
2.2.1 Sistemas de Control de Versiones Locales 31
2.2.2 Sistemas de Control de Versiones Centralizados 31
2.2.3 Sistemas de Control de Versiones Distribuidos 32
2.3 Contenedores *33*
2.3.1 Historia 34
2.3.2 Diferencias respecto a las máquinas virtuales 35
2.3.3 Diferentes tecnologías de contenedores en la actualidad 36
2.3.4 Entorno de desarrollo desplegado sobre contenedores 37

3 Tecnologías empleadas **39**
3.1 GitLab *39*
3.1.1 Historia 39
3.1.2 Instalación y configuración 40
3.1.3 Conceptos básicos 40
3.2 Git *44*
3.2.1 Historia y puntos fuertes respecto a otros VCS 44
3.2.2 Fundamentos 45
3.2.3 Instalación y configuración 46
3.2.4 Conceptos básicos 48
3.3 Docker *55*

3.3.1 Historia 55
3.3.2 Fundamentos 56
3.3.3 Instalación y configuración 57
3.3.4 Conceptos básicos 59

4 Despliegue del entorno de desarrollo 67
4.1 Imagen Docker para desarrollo 67
4.1.1 Estructura del entorno de desarrollo 67
4.1.2 Flujo de trabajo 74
4.2 Herramienta centralizada: GitLab 83
4.2.1 Configuración a nivel administrador 87
4.2.2 Flujo de trabajo a nivel de usuario 98

5 Conclusiones 117

Referencias 119

ÍNDICE DE ILUSTRACIONES

Ilustración 1 Esquema básico del escenario 24
Ilustración 2 Ciclo de vida del software 29
Ilustración 3 Sistema de control de versiones local 31
Ilustración 4 Sistema de control de versiones centralizado 32
Ilustración 5 Sistema de control de versiones distribuido 33
Ilustración 6 Diferencias en la arquitectura de Máquinas Virtuales (izq.) y Contenedores (der) 36
Ilustración 7 Distribución del uso entre las diferentes tecnologías de contenedores 36
Ilustración 8 Logo de GitLab 39
Ilustración 9 Creación de proyectos en GitLab 41
Ilustración 10 Vista principal de un proyecto 41
Ilustración 11 Tablero de incidencias 42
Ilustración 12 Revisión de código 43
Ilustración 13 Logo de Git 44
Ilustración 14 Otros VCS (arriba) VS Git (abajo) 45
Ilustración 15 Estados de los archivos en Git 46
Ilustración 16 Áreas de Git 46
Ilustración 17 Repositorio remoto 53
Ilustración 18 Rama *master* e instantánea actual 53
Ilustración 19 Rama *testing* creada desde *master*; *HEAD* apunta a *master* (git branch testing) 54
Ilustración 20 Rama *testing* creada desde *master*; *HEAD* apunta a *testing* (git checkout -b testing) 55
Ilustración 21 Divergencia de *master* y *testing* 55
Ilustración 22 Logo de Docker 56
Ilustración 23 Aplicación de ejemplo en funcionamiento 62
Ilustración 24 Esquema completo del entorno 68
Ilustración 25 Estructura del repositorio 69
Ilustración 26 Tema elegido para Zsh: avit 74
Ilustración 27 Captura de pantalla real del acceso a la imagen de desarrollo 78
Ilustración 28 Editor de texto en el equipo anfitrión 79
Ilustración 29 Catálogo de Bitnami con la búsqueda de GitLab activa 84
Ilustración 30 Configuración de la instancia 85
Ilustración 31 Panel de gestión de la instancia desplegada 86
Ilustración 32 GitLab login 86
Ilustración 33 Vista principal GitLab 87
Ilustración 34 Botones/Enlaces de acceso a la configuración 88
Ilustración 35 Página principal de configuración 88

Ilustración 36 Información del sistema 89
Ilustración 37 Logs de la aplicación e icono de acceso a las opciones 89
Ilustración 38 Página de configuración de usuarios 90
Ilustración 39 Nuevo usuario 91
Ilustración 40 Usuario creado 91
Ilustración 41 Página de configuración de usuarios con el usuario creado 92
Ilustración 42 Página de configuración de proyectos 93
Ilustración 43 Nuevo proyecto 93
Ilustración 44 Proyecto creado 94
Ilustración 45 Página de configuración de grupos 95
Ilustración 46 Nuevo grupo 96
Ilustración 47 Grupo creado 96
Ilustración 48 Página de configuración de grupos con el grupo creado 97
Ilustración 49 Configurar apariencia 97
Ilustración 50 Primer acceso de un usuario regular 98
Ilustración 51 Página principal tras el acceso 99
Ilustración 52 Crear nuevo proyecto 100
Ilustración 53 Proyecto *sample-PHP-app* creado 101
Ilustración 54 Detalle del botón "Clone" 101
Ilustración 55 Proyecto existente con Git añadido a GitLab 103
Ilustración 56 Proyecto existente sin Git añadido a GitLab 104
Ilustración 57 Configuración de GitHub 105
Ilustración 58 Crear token en GitHub 105
Ilustración 59 Crear token con permisos sobre repositorios 106
Ilustración 60 Token generado 106
Ilustración 61 Crear nuevo proyecto importado de GitHub 107
Ilustración 62 Token creado en GitHub y pegado en GitLab 107
Ilustración 63 Lista de proyectos importables 108
Ilustración 64 Proyecto importado y dirección para clonarlo 108
Ilustración 65 Crear nuevo issue con información sobre el cambio, asignación, fecha, etc. 109
Ilustración 66 Issue en *Doing* y notificaciones 110
Ilustración 67 Detalles de la tarea 110
Ilustración 68 Vista del proyecto con dos ramas 113
Ilustración 69 Estado del proyecto en la nueva rama 113
Ilustración 70 Crear merge request 114
Ilustración 71 Revisión de código 114
Ilustración 72 Estado del proyecto tras finalizar el merge request 115
Ilustración 73 Vista del fichero con el cambio y opciones para editarlo 116

ÍNDICE DE BLOQUES DE CÓDIGO

Bloque de código 1 Instalar Git 47
Bloque de código 2 Git en funcionamiento 47
Bloque de código 3 Configuración de Git 48
Bloque de código 4 Añadir y confirmar cambios 49
Bloque de código 5 Clonar repositorio 49
Bloque de código 6 Flujo de trabajo y áreas de Git 50
Bloque de código 7 Diferencial de cambios 51
Bloque de código 8 Revisar histórico de cambios 51
Bloque de código 9 Información de un commit a partir de su checksum 52
Bloque de código 10 Ramas 54
Bloque de código 11 Añadir repositorio Docker 57
Bloque de código 12 Instalar Docker desde repositorio 57
Bloque de código 13 Añadir usuario docker 58
Bloque de código 14 Docker en funcionamiento 59
Bloque de código 15 Dockerfile de ejemplo 60
Bloque de código 16 Construir imagen 61
Bloque de código 17 Ejecutar imagen 62
Bloque de código 18 Comandos ejecutados en el contenedor 63
Bloque de código 19 Terminal interactiva 63
Bloque de código 20 Terminal interactiva. Contenedor en ejecución 63
Bloque de código 21 Volúmenes 64
Bloque de código 22 Confirmar imágenes a partir de estado intermedio 65
Bloque de código 23 *Dockerfile* entorno de desarrollo 71
Bloque de código 24 Fichero *entrypoint.sh* 73
Bloque de código 25 Fichero 00*_configureZsh.sh* 74
Bloque de código 26 Desarrolladores modificando la imagen (vista de Git) 75
Bloque de código 27 Flujo de trabajo 1: clonado del repositorio de la imagen 76
Bloque de código 28 Flujo de trabajo 2: Construcción de la imagen 76
Bloque de código 29 Flujo de trabajo 3: Ejecución de la imagen 77
Bloque de código 30 Flujo de trabajo 4: Situar proyectos en el directorio adecuado 79
Bloque de código 31 Flujo de trabajo 5: Cambios realizados en local vía IDE aparecen en el contenedor 80
Bloque de código 32 Modificación del Dockerfile para quitar todos los runtimes excepto Ruby 81
Bloque de código 33 Nueva imagen a partir de *Dockerfile* modificado 82
Bloque de código 34 Flujo de trabajo 6: Personalización mediante imagen intermedia 83
Bloque de código 35 Acceso al entorno de desarrollo y comprobación de los proyectos existentes 100

Bloque de código 36 Acceso al entorno y mover a GitLab proyecto existente 102
Bloque de código 37 Mover a GitLab un proyecto existente sin Git 103
Bloque de código 38 Flujo de trabajo para hacer cambios en el proyecto 112

GLOSARIO DE TÉRMINOS

CD: Continuous Delivery (Entrega Continua)

CI: Continuous Integration (Integración Continua)

CVCS: Centralized Version Control System (Sistema de Control de Versiones Centralizado)

DVCS: Distributed Version Control System (Sistema de Control de Versiones Distribuido)

GCP: Goofle Cloud Platform

GCE: Google Cloud Engine

IDE: Integrated Development Environment (Entorno de Desarrollo Integrado)

PnP: Plug and Play (Instalación Automática)

QA: Quality Assurance (Garantía de Calidad)

VCS: Version Control System (Sistema de Control de Versiones)

1 INTRODUCCIÓN

En este capítulo se expone la necesidad de la que surge este proyecto y a la cual se le da solución. También desarrolla una serie de ventajas y desventajas que tienen en común los entornos de desarrollo. Para acabar, se realiza una breve introducción al resto de capítulos que componen este documento.

1.1 Motivación y contexto

La motivación de este proyecto es dar respuesta al deseo de un equipo de desarrollo software de actualizar sus herramientas de trabajo actuales, así como mejorar la gestión del proceso software en los proyectos desarrollados.

Actualmente se parte de una situación inicial en la que no se utiliza ninguna herramienta a nivel colaborativo para gestionar el código, cada desarrollador usa sus propios métodos para gestionar el código que está desarrollando para luego compartirlo con el resto del equipo sin usar una herramienta que englobe el trabajo de todo el equipo.

Como se verá a lo largo de esta introducción, y sin entrar aún en detalles técnicos, la implementación de un entorno de desarrollo unificado entre todos los miembros del equipo, así como disponer de una plataforma centralizada común para todos ellos donde confluye el trabajo individual para convertirse en un proyecto colaborativo, proporciona numerosas ventajas en el trabajo diario de los miembros del equipo. Esto se refleja en la calidad del proyecto y más concretamente del código que lo compone, aunque también se ve reflejado en otras etapas de la vida del proyecto software como puede ser la planificación, estimaciones o despliegues en producción.

1.2 Objetivos

Grosso modo, se pretende diseñar y configurar un entorno de desarrollo que facilite el flujo de trabajo de un equipo multidisciplinar en el que coexisten diferentes roles. Por tanto, algunas de las herramientas o funcionalidades que debe contener este entorno de desarrollo son las siguientes:

- Herramientas para el trabajo de los desarrolladores, como librerías y frameworks de los lenguajes de programación usados por el equipo o editor de texto mediante la línea de comandos.
- Sistema de control de versiones.
- Aplicación para la gestión de tareas, útil tanto para los roles técnicos como aquellos relacionados con la gestión de proyectos.
- Capacidad para almacenar los repositorios con el código.
- Herramienta de Continuous integration/Continuous Delivery (CI/CD[2]) para garantizar la calidad del software mediante la ejecución de diferentes validaciones y tests.

[2] Del inglés "Continuous integration (CI) and continuous delivery (CD)" es un conjunto de prácticas que basan su funcionamiento en la automatización de procesos, de tal manera que los cambios en el código estén disponibles de manera incremental para su despliegue lo antes posible, pasando por una serie de etapas previas como pueden ser compilación, construcción, testeo, etc.

1.3 Escenario propuesto

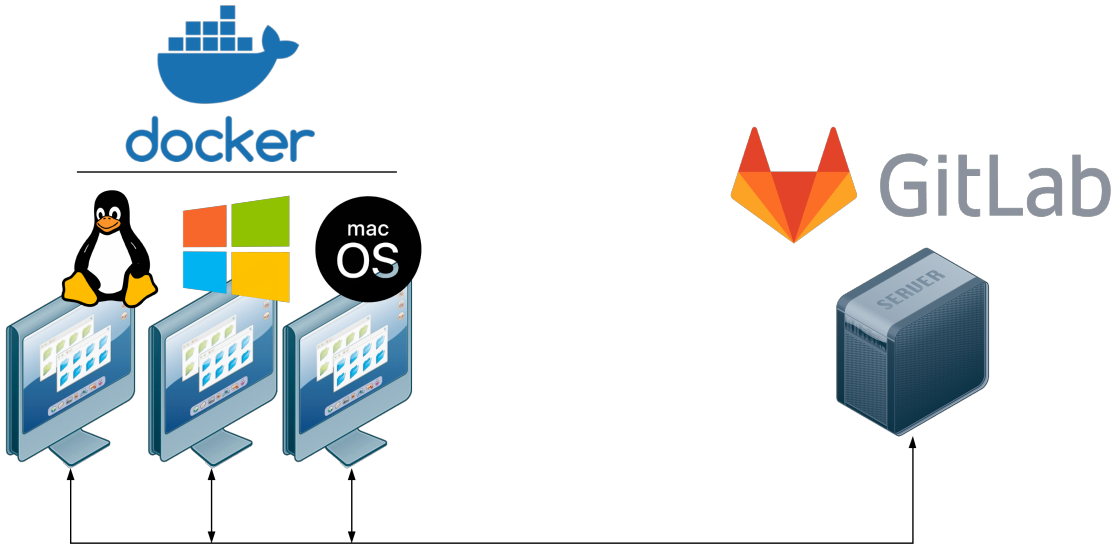

Ilustración 1 Esquema básico del escenario

Para conseguir los objetivos anteriormente mencionados se va a desplegar un escenario que consta de los siguientes elementos:

- Entorno de trabajo local que proporciona a los integrantes de un equipo de desarrollo servicios y facilidades para mejorar el flujo de trabajo diario mediante la provisión de una serie de herramientas proporcionadas de manera autocontenida en una imagen Docker que se despliega de manera sencilla mediante un contenedor.
- Herramienta de gestión de código y tareas, GitLab. Esta aplicación de código libre sirve como gestor de repositorios basado en Git, también dispone de herramientas de CI/CD, alojamiento de wikis, gestión de tareas, revisión de código y otras muchas funcionalidades en las cuales se profundizará en los siguientes capítulos.
- Sistema de gestión del código y control de versiones usando una herramienta específica para ello, Git. Este es el nexo entre el entorno de el contenedor Docker usado para el desarrollo y la herramienta de gestión del código y tareas.

1.4 Ventajas del escenario

Una vez presentado el escenario inicial, se hará hincapié en las ventajas que aporta este tipo de entornos a un equipo de desarrollo software colaborativo.

Algunas de estas ventajas surgen debido a la elección de las diferentes herramientas incluidas en la imagen de desarrollo o por la aplicación elegida para el servicio centralizado, en cambio otras están ligadas a la propia naturaleza de los contenedores y por tanto se aplican a este entorno al elegir esta solución para el despliegue del entorno de desarrollo.

En este caso se parte de un estado actual en el que no se hace uso de este tipo de herramientas y metodologías de trabajo, en otros escenarios donde el punto de partida es diferente algunas de estas ventajas podrían estar previamente cubiertas mediante otras herramientas o configuraciones.

1.4.1 Desde el punto de vista del desarrollador

- La **gestión de tareas** en un panel visual como el proporcionado por GitLab otorga al desarrollador una manera simple de conocer el estado de su trabajo actual, facilitando su organización y por tanto mejorando la productividad.
- Debido a la naturaleza de los contenedores, permiten cierta independencia respecto al sistema operativo del equipo anfitrión, lo cual otorga **portabilidad** al entorno de desarrollo. Para el desarrollador esto proporciona libertad a la hora de elegir la máquina en la que realiza su trabajo [1] [2].
- Unido al punto anterior, la distribución del código en un servidor centralizado se traduce en una mayor **flexibilidad** respecto al lugar físico de trabajo, pudiendo seguir las tareas encomendadas desde otra máquina en otra ubicación tan solo descargando la imagen del contenedor y clonando los repositorios de código [1] [2].
- El propio desarrollador ve enriquecido su conocimiento gracias a las **revisiones de código** recibidas el resto de los compañeros, lo cual se traduce en una mayor calidad del código y también hace que el desarrollador mejore sus habilidades aprendiendo de estas revisiones.
- La facilidad para restaurar la imagen Docker y devolverla a su estado inicial proporciona una gran ventaja con respecto a otros entornos de desarrollo como por ejemplo el desarrollo directamente en el equipo anfitrión. Así, en caso de necesitar **restaurar** el entorno a su estado inicial es tan fácil como ejecutar un par de comandos [1] [2]; de otra manera, si el desarrollo es en un equipo personal y se llega al punto de tener que eliminar ciertos paquetes instalados, sus dependencias, etc. es una tarea más tediosa que no pocas veces acaba en un formateo de la máquina. También es posible ir creando imágenes intermedias a partir de la imagen inicial, de esta manera, usando un sistema de etiquetado correcto, no solo será posible restaurar el sistema al estado inicial, sino a cualquier estado intermedio si el desarrollador considera que ha hecho cambios notables en su entorno de desarrollo.
- Gracias al sistema de control de versiones y al alojamiento del código en un servidor centralizado el desarrollador tiene siempre la seguridad de tener diferentes formas de restaurar el código. Lo cual se traduce en mayor seguridad a la hora de desarrollar y realizar cambios, pudiendo restaurar versiones anteriores en el caso de introducir un error.
- Como se verá en el siguiente apartado relativo a las ventajas desde el punto de vista del proyecto, uno de los puntos fuertes es la homogeneización del entorno entre los diferentes desarrolladores. Esto podría contravenir las necesidades individuales de cada miembro del equipo, sin embargo, el entorno diseñado en este proyecto permite al desarrollador **personalizar** su contenedor mediante el uso de scripts de inicialización, respondiendo tanto a necesidades personales como del proyecto, puesto que no todos los miembros de un equipo multidisciplinar necesitan las mismas herramientas.
- El entorno de desarrollo contiene todo lo necesario para empezar a trabajar de manera inmediata, como se ha visto en el punto anterior sobre personalización, quizás no sea así en la primera iteración, pero sí en las siguientes. Siguiendo así el concepto **plug and play** (**PnP**) donde el usuario no debería preocuparse acerca de cualquier tipo de configuración inicial requerida. El entorno trae instalado los runtimes de los lenguajes de programación más habituales, así como una terminal orientada al desarrollo (Zsh) que como se verá en los siguiente capítulos, otorga una serie de ventajas respecto a otras opciones como sh o bash.
- Este enfoque deja a elección del desarrollador el uso de su **IDE o editor de texto favorito**, solventando así uno de los temas más controvertidos cuando se intenta dotar de un sistema de trabajo común y estandarizado para un grupo diverso de trabajadores. Esto es posible gracias al uso de volúmenes en los contenedores, básicamente se monta un directorio que es compartido entre el equipo anfitrión y el contenedor, haciendo transparente durante la edición la existencia del contenedor.

1.4.2 Desde el punto de vista del proyecto

- La confluencia de la gestión de tareas y gestión del código en una misma aplicación hace que se obtenga una visión de más alto nivel sobre el estado del proyecto. De igual manera es posible adjuntar

código a tareas específicas, solicitar la revisión de este código al resto de miembros del equipo trabajando en esa tarea, etc. De esta manera se aúnan dos partes importantes de cualquier proyecto software que no siempre van de la mano.

- Una ventaja importante para un proyecto software es la **reproducibilidad**, es decir, poder reproducir el mismo entorno que se tenía en un momento concreto cuando se detectó algún incidente. Muchas veces salen a la luz ciertas situaciones en las que aparece cualquier tipo de error y al cabo de un tiempo desaparece sin saber el motivo ni si volverá a ocurrir. Usando este tipo de entornos, existe control sobre la imagen usada en cualquier momento, de tal manera que se puede volver al estado anterior y reproducir el mismo escenario.
- Otro punto positivo es la **homogeneización** del sistema entre los diferentes desarrolladores evitando así inconsistencias debido a versiones de herramientas, sistemas operativos, librerías instaladas, variables de entorno, etc. Lo cual se traduce en una mayor consistencia a la hora de desarrollar el código, realizar tests, etc.
- Al utilizar un sistema de control de versiones distribuidos, como es Git, junto a un servidor de repositorios, como es GitLab, se dispondrá de copias de seguridad de las distintas versiones del código no solo en el servidor sino en uno o varios equipos de los desarrolladores.
- Con una correcta configuración de Git tanto en el cliente como en el servidor, se proporciona cierto nivel de **seguridad** sobre la autoría del código, es decir, se tiene control en todo momento sobre quién hace cada cambio evitando así la introducción intencionada de bugs u otras prácticas maliciosas.
- Un sistema de este tipo se puede dotar con diferentes formas de **automatización**, tanto a nivel del contenedor de desarrollo como del servidor central [3]. Siendo las herramientas de CI/CD las más avanzadas para este tipo de tareas.
- Es una solución **escalable**, es decir, no depende del número de desarrolladores trabajando en el producto.

1.5 Desventajas del escenario

De la misma manera que en el apartado anterior se han listado una serie de ventajas que presenta este tipo de entornos de trabajo, el mismo no está exento de una serie de desventajas o puntos en contra, los cuales se listan a continuación, pudiendo no ser los únicos:

- La **curva de aprendizaje** puede ser significativa no por la dificultad aislada de ninguna de las herramientas usadas, puesto que todas ellas son opciones comunes, si no por el hecho de usar todas ellas de manera conjunta en un entorno que engloba diferentes características.
- El uso de las tecnologías de contenedores y repositorios distribuidos puede generar cierta **desconfianza** sobretodo relativa a la localización del código y el miedo a perder el trabajo hecho, aunque como se verá más adelante, conociendo la tecnología se confirmará que esto no es así puesto que los datos están replicados en el equipo anfitrión.
- Este sistema tiene una fuerte dependencia de una **conexión a internet**. La primera vez que se va a hacer uso del entorno se necesita clonar los repositorios y hacer pull de la imagen del contenedor. En sucesivas ejecuciones, tanto la imagen como el código están en el equipo anfitrión y no hace falta dicha conexión, pero sí es requerida para sincronizar el código con el repositorio remoto.
- Como se ha mencionado en la sección de ventajas, uno de los puntos fuertes es el hecho de poder restaurar el sistema de una manera sencilla, esto entra en conflicto con el hecho de **persistir** los paquetes o herramientas instalados en el contenedor, de tal manera que todo aquello que no vaya incluido en la imagen o en los scripts de inicialización y se haya instalado una vez desplegado el contenedor se perderá en las sucesivas ejecuciones. Este problema puede ser solventado mediante la creación de imágenes intermedias, lo cual se explica en el capítulo 4.
- Ausencia de entorno gráfico en la solución propuesta como contenedor. Aunque como se ha dicho, el desarrollador puede usar el IDE o editor de texto gráfico que tenga instalado en el equipo anfitrión, el

contenedor no tiene interfaz gráfica puesto que no es un Sistema Operativo al uso, por tanto, la interacción será mediante línea de comando.

En los próximos capítulos se entrará en detalles sobre las tecnologías usadas a lo largo del proyecto, encontrando así justificación técnica y de diseño a los puntos anteriormente desarrollados.

1.6 Estructura de la memoria

La finalidad de este documento es explicar la solución implementada para solventar la necesidad que se ha planteado. Para ello, se ha divido en una serie de capítulos en los que se trata de documentar el proceso seguido; desde las ideas iniciales y búsqueda de las tecnologías que satisfacen las necesidades planteadas, hasta las conclusiones a las cuales se ha llegado.

En primer lugar, aparece el capítulo actual, el cual hace una introducción de la necesidad que pretende solventar este proyecto, así como una serie de ventajas y desventajas que tienen en común todas las soluciones basadas en este tipo de escenarios.

En el segundo capítulo se plantea el estado de las tecnologías relacionadas con la solución propuesta, lo cual ha sido fruto de la investigación inicial llevada a cabo para determinar las mejores opciones a la hora de implementar la solución que da respuesta a esta necesidad.

En el capítulo número tres se exponen las tecnologías concretas que se han usado en la implementación del proyecto, sin entrar en detalles de implantación en este caso concreto, se aborda desde una perspectiva genérica.

El cuarto capítulo trata todo lo relativo a la implementación del proyecto, sus características y una explicación de las partes de las que consta de manera detallada, así como de su uso y configuración inicial.

En el quinto y último capítulo, se exponen una serie de conclusiones sacadas tras la realización de este proyecto, así como posibles mejoras realizables en un futuro.

2 ESTADO DE LA TÉCNICA

Durante este capítulo se plantea el estado de las tecnologías claves que conforman este proyecto. Se puede decir que este capítulo es el fruto de la investigación inicial llevada a cabo para determinar las mejores opciones a la hora de implementar la solución.

El entorno de desarrollo propuesto como solución tiene tres pilares fundamentales, los cuales son: El contenedor que se usa para desplegar las herramientas de desarrollo, la aplicación de gestión del ciclo de vida del software y una herramienta de control de versiones encargada, entre otras cosas, de distribuir el código y sincronizarlo entre el servidor y los clientes.

2.1 Gestión del ciclo de vida del software

La gestión del ciclo de vida del software es el proceso que se sigue para diseñar, desarrollar, entregar, monitorizar, evolucionar y retirar el software, desde la concepción de una idea hasta la obsolescencia del producto. Este proceso consta de diferentes etapas desde su nacimiento hasta su desaparición [4].

No existe una categorización clara de las diferentes etapas que conforman el ciclo de vida de un producto software, aunque generalmente se coincide en una serie de fases comunes en la mayoría de los proyectos y metodologías de desarrollo [5]:

- **Evaluación de necesidades**: es la recopilación y comunicación con el cliente sobre las necesidades y limitaciones que debe soportar el software en cuestión.
- **Diseño o especificación**: se puede dividir en diferentes fases según el nivel de granularidad deseado, se suele comenzar con una especificación global de las funcionalidades del software para luego entrar en una especificación más detallada en la que se define la arquitectura del sistema en cuestión y por último terminar con las especificaciones técnicas [4] [6].
- **Implementación**: esta es la fase de desarrollo. Los desarrolladores traducirán al código las funcionalidades descritas en las etapas anteriores.
- **Pruebas unitarias**: las pruebas unitarias se usan para verificar que cada unidad de código, al ejecutarse, actúe de acuerdo con las especificaciones proporcionadas en las primeras fases. Idealmente, los programadores escriben las pruebas durante la fase de implementación, por lo tanto, es una fase que se realiza de manera simultánea o consecutiva con la fase anterior, estas fases no están desacopladas [5].
- **Integración**: todos los elementos desarrollados se integran para evaluar la correcta interacción y el buen funcionamiento de estos en conjunto. Estos controles se realizan mediante pruebas de integración, que al igual que las pruebas unitarias es deseable que se ejecuten junto a la implementación.
- **Aceptación**: el cliente evalúa la adecuación de la aplicación desarrollada con las especificaciones definidas, dicha evaluación se suele realizar mediante unas pruebas de aceptación [4] [6].

Aunque un equipo de desarrollo puede participar activamente en cualquiera de las fases anteriormente descritas, lo normal es que se centren en las fases de Implementación, Pruebas unitarias e Integración.

Para ello el equipo de desarrollo recibe una serie de tareas basadas en los datos obtenidos tras la Evaluación de necesidades y el Diseño o especificación, de tal manera que la tarea del equipo de desarrollo sea realizar el software que cumpla con lo recogido en los documentos de diseño.

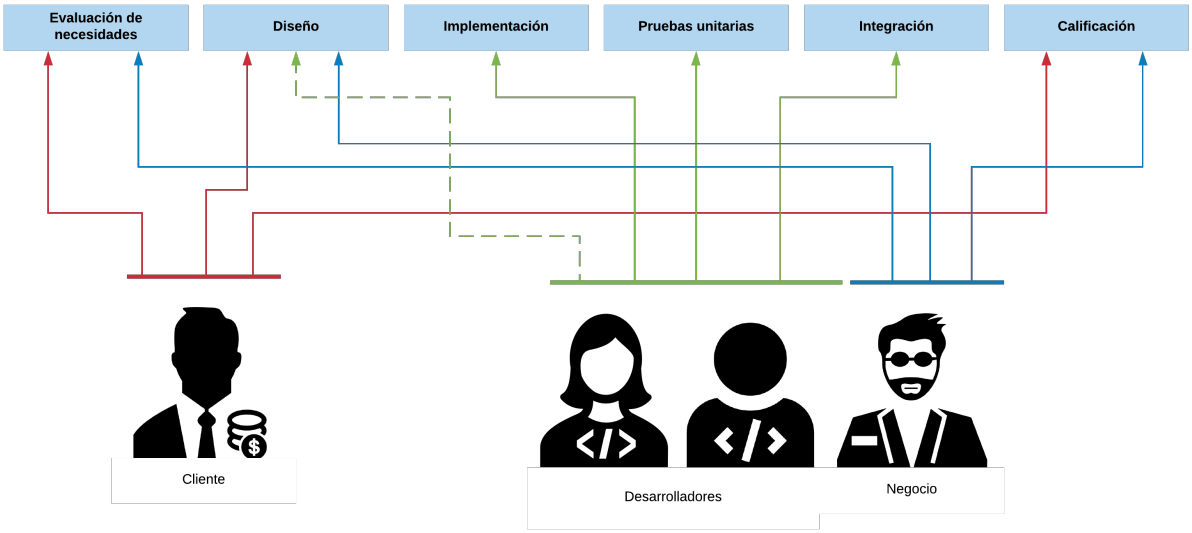

Ilustración 2 Ciclo de vida del software

En base a los acuerdos o contratos establecidos con el cliente, el ciclo de vida suele continuar tras la aceptación o entrega del producto, puesto que se requiere soporte a los usuarios, corregir bugs, actualizaciones, etc. Tareas que, nuevamente, suelen recaer en la figura de los desarrolladores.

Puesto que este proyecto pretende dar respuesta a una necesidad surgida de un equipo de desarrollo (y no de una compañía completa), se focalizará en aquellas fases que competen a los desarrolladores.

La información generada tras la Evaluación de necesidades y el Diseño suele estar en forma de documentos, los cuales el jefe de proyecto, el mánager del equipo o la persona que tenga el rol de gestionar el equipo de desarrolladores será el encargado de "traducir" la información de estos documentos a tareas que los desarrolladores ejecuten.

Estas tareas deben ser correctamente evaluadas y estimadas para ajustarlas a la planificación temporal del equipo de desarrollo, de igual manera, dichas tareas pueden pasar por diferentes fases (futuro, en progreso, esperando revisión, bloqueada, hecha, etc.) dentro de la fase de implementación; esto dependerá de la metodología de trabajo seguida por el equipo (Scrum, Extreme Programming, Agile, etc.) [7].

En este proyecto se ha dotado al entorno de desarrollo con una aplicación que soluciona las diferentes necesidades del equipo no solo en cuanto a las tareas de desarrollo entendidas como tal, si no también en otras fases del ciclo de vida del software. El trabajo del equipo de desarrollo suele continuar tras la entrega, bien para corregir errores o incluir nuevas funcionalidades. Como se verá en los siguientes capítulos, la aplicación elegida pretende dar respuesta a estas necesidades.

2.1.1 Comparativa de herramientas

Existen muchas herramientas para la gestión del ciclo de vida del softwate. Partiendo de la base de un catálogo de unas 40 aplicaciones en un primer cribado se han descartado 18 debido a su licencia comercial o privativa, puesto que se está interesado en aplicaciones de uso gratuito. De la misma manera se han descartado otras 16 aplicaciones por diversos motivos como pueden ser: no soportar control de versiones, proyecto abandonado, presencia insuficiente, etc.

Finalmente, quedan 5 posibles candidatos para ser usados como aplicación centralizada que sirva principalmente para alojar y gestionar los repositorios de código y dar soporte a la gestión de tareas. En la comparativa para decidir la solución también se tienen en cuenta otras características que sean de utilidad en las diferentes fases del proceso de gestión integral del software [8].

Solución / Característica	**Bugzilla** [9]	**MantisBT** [10]	**GitLab** [11]	**Phabricator** [12]	**Redmine** [13]
Creador	Mozilla Foundation	Open source contributors	GitLab Inc.	Facebook	Jean-Philippe Lang
Licencia	MPL	GPL	MIT	Apache	GPLv2
Implementación	Perl	PHP	Ruby, Go y Vue.js	PHP	Ruby
Lanzamiento	1998	2000	2011	2010	2006
Wikis	Si	Si	Si	Si	Si
Interfaz web	Si	Si	Si	Si	Si
CLI	Si	No	Si	Si	Si
Email	Si	Si	Si	Si	Si
Integración Git	Si	Si	Si	Si	Si
Revisión de código	Externo	No	Si	Si	Plugin
CI/CD	No	Externo	Si	Si	No

Tabla 1 Comparativa herramientas gestión software

A tenor de la comparativa realizada, tanto Phabricator como GitLab se amoldan a las necesidades de este proyecto, para decidir cuál de las opciones es la que finalmente se utiliza como pieza central de este entorno de desarrollo se ha profundizado en otros aspectos como la adopción que tiene la solución, número de contribuciones en GitHub, periodicidad de las raleases, etc. Finalmente, basándose en todos estos aspectos se ha decidido usar GitLab como la aplicación centralizada para gestionar el código y las tareas del equipo de desarrolladores.

2.2 Control de versiones

Un Sistema de Control de Versiones (VCS por sus siglas en inglés), es un sistema que registra los cambios realizados en un conjunto de archivos a lo largo del tiempo, de modo que se puedan recuperar versiones específicas en cualquier momento. Dicho sistema permite regresar a versiones anteriores de cualquier archivo o incluso del proyecto entero, permite comparar cambios a lo largo del tiempo, ver quién realizó un cambio concreto, unificar el trabajo de varios desarrolladores ayudando a resolver los conflictos que puedan aparecer, etc. [14] [15].

Todo lo anterior y otra serie de ventajas, hacen que el desarrollo de software en equipos distribuidos sea una tarea más cómoda y menos propensa a errores, y en caso de haberlos permite una fácil localización y regresión a versiones del código estable [14]. Especialmente útil si hay varias personas trabajando en el mismo repositorio de código de manera simultánea. Por todo lo anterior es una pieza clave en este proyecto [16] [17].

A continuación, se verán los diferentes enfoques existentes para realizar este control de versiones:

2.2.1 Sistemas de Control de Versiones Locales

El método tradicional de control de versiones es copiar y pegar los archivos a otra localización, o incluso a la misma, cambiando los nombres de estos; de esta manera se consigue un versionado el cual puede ser más "funcional" si además de cambiar el nombre de los ficheros se añade al mismo una fecha y/o autor.

Como cabría esperar, este método es relativamente común debido a su sencillez, pero también es muy propenso a errores puesto que es fácil cometer errores con el versionado, modificar o eliminar el archivo que no se deseaba, etc.

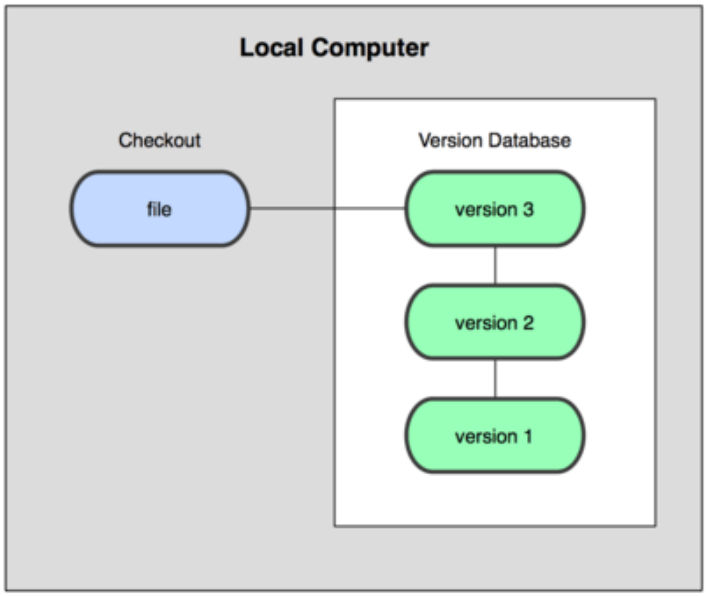

Ilustración 3 Sistema de control de versiones local

Para afrontar este problema los programadores desarrollaron hace tiempo VCS locales que contenían una simple base de datos en la que se llevaba el registro de todos los cambios realizados a los archivos. Cuando era necesario revisar o modificar una versión anterior de algún archivo, se usaba el sistema de control de versiones en vez de acceder directamente al archivo, de esta manera en cualquier momento solo se tenia una copia de los ficheros, eliminando la posibilidad de confundir o eliminar versiones [18].

Este método no puede ser usado para dar solución al problema planteado en este proyecto puesto que se limita al ámbito local, para ello cada desarrollador debería seguir su propio control de versiones (usando las mismas convenciones al ser posible) y posteriormente compartir y poner en común el código usando otro método de transferencia de archivos, como podría ser el email o carpetas compartidas de almacenamiento en la nube. Aunque es factible, se verá a continuación que hay mejores soluciones.

2.2.2 Sistemas de Control de Versiones Centralizados

Este método (CVCS por sus siglas en inglés) surge para satisfacer la necesidad de poner en común el trabajo realizado por diferentes miembros de un mismo equipo (o incluso entre miembros de diferentes equipos y/o compañías) [18]. Estos sistemas tienen un único servidor (a no ser que el administrador haya configurado el sistema con redundancia) que contiene todos los archivos versionados, y varios clientes que descargan (pull) y suben (push) los archivos desde/hacia ese lugar central. Este ha sido el estándar para el control de versiones por muchos años.

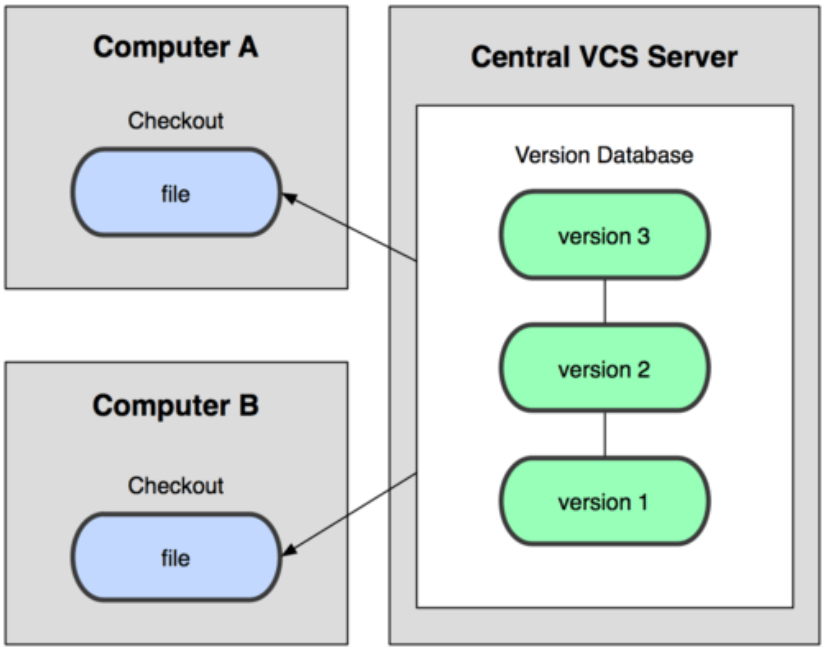

Ilustración 4 Sistema de control de versiones centralizado

Este segundo método aún tiene algunas desventajas que hacen que no sea la solución idónea para este proyecto. La más importante es el punto único de fallo que representa el servidor centralizado, puesto que a priori, y a no ser que el administrador lo dote de algún tipo de replicación y copias de seguridad, si ese servidor cae o sufre una pérdida de datos se producirá una interrupción total del servicio.

2.2.3 Sistemas de Control de Versiones Distribuidos

Por último, este método (DVCS por sus siglas en inglés) ofrece soluciones para los problemas descritos anteriormente y por los cuales han sido descartados los métodos precedentes. En los sistemas de este tipo, se utiliza también un servidor de repositorio, en cambio los clientes no solo descargan la última versión, sino que replican compeletamente el repositorio [18] [19]. De esta manera, si se pierden datos en el servidor, cualquiera de los repositorios disponibles en los clientes puede ser copiado al servidor con el fin de restaurarlo. Cada repositorio local es una copia de todos los datos de este repositorio disponibles en el servidor, aunque las acciones de push y pull se siguen realizando contra el servidor.

El servidor puede disponer de varios repositorios de código, por lo que puede alojar código de diferentes equipos e incluso compañías. Obviamente los clientes tan solo clonan los repositorios que el desarrollador crea conveniente, no realiza un clon de todo el servidor.

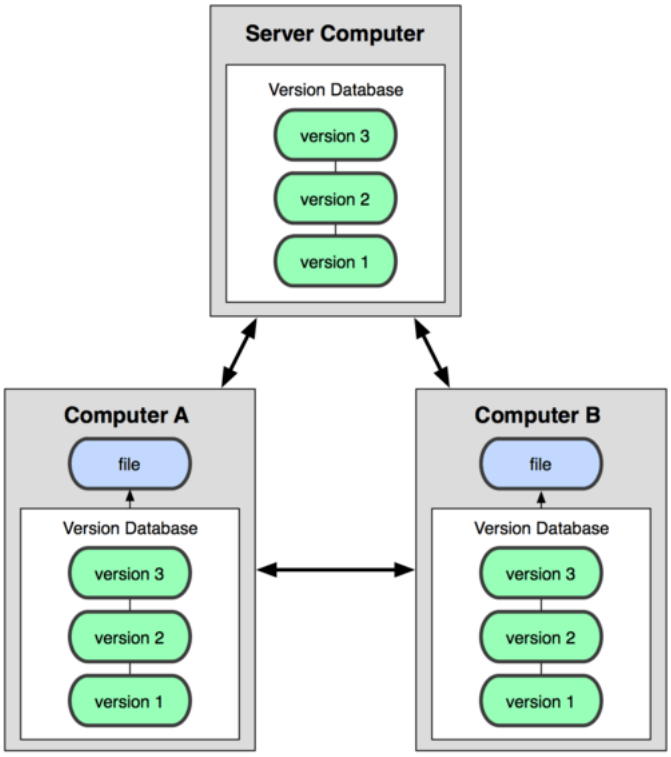

Ilustración 5 Sistema de control de versiones distribuido

En un equipo que trabaja de manera distribuida, cada cual necesita controlar el versionado de los ficheros con los que está trabajando, pero teniendo que poner en común los avances realizados de manera periódica para que los otros miembros incorporen los cambios recientes a su código.

Por los motivos expuestos anteriormente, para este proyecto tiene más sentido usar un sistema de control de versiones distribuido. GitLab cumple las funciones de servidor de repositorios, teniendo también la ventaja de que el código de los proyectos de este equipo no está alojado en un repositorio público o proveedor externo.

Existen diferentes soluciones de Sistemas de Control de Versiones, entre las que destacan CVS, Git, Subversion, Mercurial y Baazar [20]. Para este proyecto se ha elegido Git debido a que implementa control de versiones distribuido, que, como se ha visto anteriormente, es el sistema que mejor se adapta a las necesidades del proyecto [21]. De la misma forma, es el sistema de facto en la mayoría de los proyectos software, lo cual hace que sea un sistema conocido por la mayoría de los desarrolladores, favoreciendo también la relación con los proyectos open source, tanto para contribuir con proyectos externos como para recibir contribuciones de la comunidad [16].

2.3 Contenedores

No se puede hablar de contenedores sin antes introducir el concepto de virtualización, aunque sea de una manera muy superficial. Básicamente la virtualización nace para aprovechar los recursos sobrantes de una máquina. El hardware disponible cada vez era más potente y las aplicaciones software no necesitaban ocupar toda esa capacidad física. Por este motivo se crearon "recursos virtuales", de tal manera que se simula un hardware dedicado donde se ejecuta el software; pero realmente el hardware es compartido (de una manera más o menos aislada según la tecnología de virtualización usada), permitiendo así que múltiples sistemas operativos puedan ser ejecutados al mismo tiempo, cada uno usando una fracción de los recursos hardware disponibles. Cada sistema operativo gestiona sus aplicaciones de manera independiente, tal y como lo haría si fueran ejecutados en máquinas independientes.

Como se ha mencionado anteriormente, existen diversas tecnologías de virtualización, siendo los contenedores una de ellas. Concretamente se puede decir que un contenedor es un entorno virtual que se ejecuta sobre el kernel del sistema operativo anfitrión, del cual utiliza ciertos recursos básicos comunes en los diferentes

sistemas operativos soportados. Se incluyen en el propio contenedor aquellos recursos más específicos, lo cual permite su ejecución en cualquier sistema operativo. De esta manera no despliega un sistema operativo completo, solo aquellos recursos que no son comunes entre los diferentes sistemas operativos.

Con respecto al uso del hardware, los contenedores tienen acceso al hardware físico del equipo anfitrión, por tanto, no requieren la virtualización de estos recursos, accediendo directamente a través de las librerías del sistema operativo.

2.3.1 Historia

Se tiende a catalogar los contenedores y las máquinas virtuales como dos tecnologías que ahora mismo conviven, pero con diferente futuro, mientras que las máquinas virtuales son una tecnología en desuso, los contenedores suponen el presente y futuro de la virtualización. Esto no es del todo cierto, ni las máquinas virtuales están en decadencia, ni los contenedores son una tecnología novedosa, aunque ha ido evolucionando con el paso de los años, los conceptos y bases de los contenedores que se conocen actualmente se plantearon hace unas décadas.

Durante el desarrollo de Unix 7 en el año 1979 se introdujo el sistema *chroot*, una operación que cambia el directorio raíz aparente para el proceso en ejecución actual y sus hijos [22]. Este avance fue el inicio del aislamiento de los procesos dentro de una máquina y consistía simplemente en separar el acceso a archivos para cada proceso. Finalmente, *chroot* fue añadido a BSD [3]en 1982 [23].

Entorno al año 2000, para satisfacer las diferentes necesidades de sus clientes, un pequeño proveedor de hosting compartido presentó "FreeBSD Jails" que conseguía lograr una fuerte separación entre sus servicios y los de sus clientes principalmente por seguridad y facilidad de administración, de esta manera se concentraban y separaban los servicios que eran propios de la infraestructura del hosting y la de sus clientes [23] [24].

FreeBSD Jails permite particionar un servidor Unix basado en este sistema operativo. Cada unidad de virtualización se llama "cárcel". Las cárceles tienen su propio usuario root y derechos de acceso, pueden usar la infraestructura de virtualización del subsistema de red o compartir redes existentes, lo cual permite a los administradores separar un sistema informático FreeBSD en varios sistemas independientes, con la capacidad de asignar una dirección IP para cada sistema y configuración [23] [24].

De manera prácticamente paralela a FreeBSD Jails, nace Linux VServer, un nuevo sistema de virtualización que puede aislar los recursos (sistemas de archivos, direcciones de red, memoria). Fue añadido directamente al kernel de Linux en el año 2001 [25]. Una vez alcanzado un nivel de virtualización similar al que se conocen hoy en día, se fueron lanzando diferentes soluciones como la de Open VZ (Open Virtuzzo) o Solaris Oracle [23].

En el año 2006 entra en juego Google y su herramienta Process Containers que fue diseñada para limitar y aislar los accesos a recursos de la máquina como CPU, memoria, I/O de disco, red, etc., por parte de un grupo de procesos. Fue renombrado "Grupos de control (*cgroups*)" en 2007 y finalmente se fusionó con el kernel de Linux [23] [26].

Unos años más tarde nace LXC (LinuX Containers) que basado en los citados *cgroups* y namespaces de Linux, consiguió la primera implementación estable del gestor de contenedores Linux [23] [26] [27].

En 2011 y 2013 surgen diversas tecnologías como Warden y LMCTFY (Let Me Contain That For You), el primero desarrolló el primer modelo cliente-servidor para administrar contenedores distribuidos en diferentes equipos [28]. LMCTFY hacía que las aplicaciones propiamente dichas tuvieran capacidad para controlar el contenedor, administrando sus propios subcontenedores o contenedores hijos [23] [29].

Fue en 2013 cuando surgió Docker y los contenedores explotaron en popularidad. El crecimiento de Docker y el uso de contenedores van de la mano [30]. Docker usó LXC en sus etapas iniciales y más tarde fue reemplazado por libcontainer. Durante estos últimos años han surgido otras alternativas a Docker, aunque sigue siendo con diferencia la solución más usada (83% en 2018) [23] [30].

[3] Berkeley Software Distribution (BSD) era un sistema operativo basado en Research Unix, desarrollado y distribuido por el Computer Systems Research Group (CSRG) de la Universidad de California, Berkeley. Hoy en día, "BSD" a menudo hace referencia a sus descendientes, entre los que se encuentran FreeBSD, OpenBSD, NetBSD o DragonFly BSD.

En los últimos años las plataformas para orquestar contenedores como docker-compose, Docker Swarm, Rancher o Kubernetes han cogido una gran fuerza en el mercado, especialmente para hacer uso de los contenedores en producción. Estas herramientas surgen para automatizar el despliegue, la gestión y el escalado de las aplicaciones basadas en contenedores. En entornos de producción no se habla de un solo contenedor por aplicación, si no que se tienen diferentes contenedores cada uno proporcionando un servicio diferente (base de datos, servidor web, métricas, la propia aplicación, etc.). Estos contenedores atienden una demanda determinada que tiene que ser satisfecha por unos recursos los cuales se tienen que escalar, actualizar, etc. sin repercutir en el usuario de la aplicación [31].

Todas estas tareas de gestión o mantenimiento son altamente manuales y propensas a errores, por tanto, el uso de estas herramientas hace que estos errores se reduzcan, lo cual se traduce a su vez en una reducción, o la eliminación total, del tiempo de inhabilitación del servicio. En este proyecto no se usa ninguna de ellas puesto que, aunque hay diferentes contenedores ejecutándose, son independientes unos de otros y no necesitan un sistema de orquestación.

2.3.2 Diferencias respecto a las máquinas virtuales

Tradicionalmente y durante los últimos años, las máquinas virtuales han sido la tecnología más extendida de virtualización. Una máquina virtual es un sistema operativo completo que funciona de manera aislada sobre otro sistema operativo gracias a un software especializado llamado hipervisor. El hipervisor expone los recursos hardware que usa normalmente el sistema operativo anfitrión para que sean usados por el sistema operativo de la máquina virtual. El sistema operativo de la máquina virtual cree en todo momento que está usando una infraestructura dedicada, es decir, a ojos del sistema operativo, tanto el anfitrión como la máquina virtual creen que están usando los recursos físicos de manera dedicada.

Los contenedores actúan de manera diferente, en vez de virtualizar un sistema operativo completo hacen uso de los recursos del equipo anfitrión. Al fin y al cabo, un contenedor se ejecuta de manera nativa en Linux, compartiendo el kernel del equipo anfitrión con el resto de los contenedores. Para el equipo anfitrión es un proceso ligero que no consume más memoria que cualquier otro ejecutable que se pueda lanzar en el sistema [32] [33]. En cambio, la máquina virtual ejecuta un sistema operativo completo con acceso a los recursos del equipo anfitrión, en la mayoría de los casos esto se traduce en un sistema sobredimensionado, en el que la máquina virtual captura más recursos de los que las aplicaciones desplegadas necesitan [33].

Una de las principales diferencias en términos de diseño con respecto a las máquinas virtuales es precisamente lo descrito en el párrafo anterior, mientras que la máquina virtual necesita un hipervisor, la virtualización de la BIOS y el hardware para ejecutar un sistema operativo completo; el contenedor opera usando el kernel del anfitrión, sin requerir hardware dedicado. Durante la provisión de una máquina virtual hay que indicar previamente los recursos que se van a asignar (número de cores, cantidad de RAM, espacio de disco, etc.) y estos recursos estarán “secuestrados” por la máquina virtual, aunque no se usen en su totalidad. En el caso de los contenedores, estos recursos se ajustan de manera dinámica en función de las necesidades del momento [1] [33].

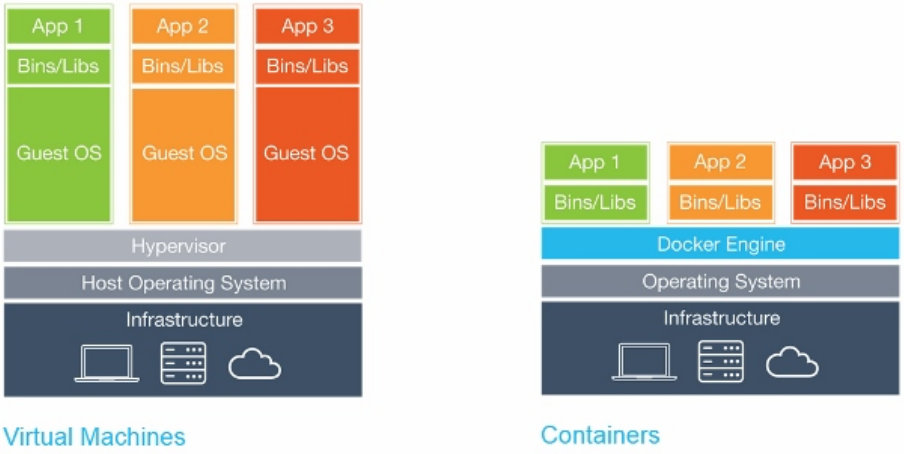

Ilustración 6 Diferencias en la arquitectura de Máquinas Virtuales (izq.) y Contenedores (der)

El hecho de que un contenedor no necesite ejecutar un sistema operativo completamente funcional, solo las librerías y/o paquetes necesarios para correr el servicio deseado, hace que el tamaño de las imágenes sea inferior respecto a las máquinas virtuales; de igual forma el tiempo de despliegue es menor para un contenedor que para una máquina virtual [1].

Una diferencia muy importante en lo relacionado con este proyecto es el hecho de que a nivel de recursos en la máquina que ejerce de anfitrión, es decir, el equipo del desarrollador, los contenedores son más ligeros al no tener que ejecutar un entorno de virtualización basado en hipervisor. El desarrollador usará su equipo de manera habitual, por tanto, aparte del entorno de desarrollo basado en contenedores, habrá otras herramientas en uso como pueden ser el navegador web, un IDE o editor de texto, aplicaciones de mensajería, etc. Si la solución propuesta no fuera respetuosa con los recursos del equipo anfitrión, todas estas tareas cotidianas que realiza el desarrollador en su equipo se verían afectadas por la escasez de recursos en el sistema.

En temas de seguridad, el aislamiento que se consigue con una máquina virtual ejecutando su propio sistema operativo no se llega a conseguir con un contenedor donde el kernel compartido entre los diferentes contenedores y el propio equipo anfitrión provoca diversas vulnerabilidades, principalmente relacionadas con la escalada de privilegios [33].

2.3.3 Diferentes tecnologías de contenedores en la actualidad

Como se mencionó al inicio de este capítulo, Docker es la tecnología más usada con diferencia (83% en 2018); pero no la única, de hecho, desde el año 2017 donde Docker copaba el 99% del mercado, la adopción de otras tecnologías ha ido en aumento [30].

Ilustración 7 Distribución del uso entre las diferentes tecnologías de contenedores

Utilizando estos datos de uso en el ultimo año, se hará una breve comparativa entre las tres soluciones más usadas:

Docker

Docker es un conjunto de productos acoplados que utilizan la virtualización a nivel de sistema operativo para desarrollar y desplegar software empaquetado, como se ha visto anteriormente, estos paquetes de software son lo que se conoce como contenedores. El software que aloja y gestiona los contenedores se llama Docker Engine [34] [35].

De manera simplificada se puede describir el funcionamiento usando los siguientes elementos. El demonio Docker, llamado *dockerd*, es un proceso persistente que administra los contenedores y maneja los diferentes objetos (volúmenes, puertos, imágenes, servicios, etc.). El demonio escucha las solicitudes enviadas a través de la API de Docker Engine. El programa cliente Docker, llamado *docker*, proporciona una interfaz de línea de comandos que permite a los usuarios interactuar con los diferentes objetos [2] [36].

Otra pieza fundamental son los diferentes registros (quay, Docker Hub, gcr, etc.). Un registro de Docker es un repositorio de imágenes de contenedores donde los clientes de Docker se conectan para descargar ("pull") imágenes que luego serán ejecutadas o cargar ("push") las imágenes que han construido localmente.

Docker proporciona sus propias herramientas de orquestación como son Docker Swarm y docker-compose; también juega a su favor el hecho de que pueda ser ejecutado en sistemas operativos basados en Linux, Windows y macOS.

CoreOS RKT

"Rocket" (o *rkt*) es una CLI[4] que permite ejecutar aplicaciones desplegadas en contenedores. Se hace énfasis en el concepto de que es una tecnología orientada a aplicaciones en producción puesto que los desarrolladores han apostado fuertemente por mejorar uno de los puntos débiles de Docker, la seguridad [37].

RKT implementa un formato de imágenes abierto y estándar, App Container (*appc*), pero también puede ejecutar otras imágenes, como las creadas usando el formato propietario de Docker.

Una de las ventajas de RKT es su orientación hacia la comunidad y el software libre, en contraposición a Docker que es un producto privativo de Docker Inc., aunque tengan gran cantidad de recursos disponibles para la comunidad y los usuarios [38].

Rocket está disponible en la mayoría de las distribuciones Linux, pero no así en otros sistemas operativos como Windows o macOS.

Mesos Containerizer

Mesos nace como un sistema de orquestación de contenedores como puede ser Kubernetes o Docker Swarm, la particularidad radica en que más allá de soportar las tecnologías de contenedores existentes, como por ejemplo Docker, también ha desarrollado su propia tecnología de contenedores llamada Mesos Containerizer [39].

La tecnología de contenedores usada de manera nativa por Mesos es Mesos Containerizer, lo cual permite controlar las tareas sin necesidad de recurrir a otra tecnología externa. También permite tener un control más minucioso del aislamiento que sufren los contenedores puesto que permite interactuar con la tecnología nativa de Linux que lo hace posible (*cgroups*/*namespaces*). Mesos Containerizer permite controlar o limitar recursos al estilo de las máquinas virtuales, como puede ser limitar el uso del disco en un contenedor concreto [40].

De igual forma, Mesos permite usar varias tecnologías a la vez, lo cual es útil para testear si una aplicación funciona en diferentes plataformas de contenedores.

2.3.4 Entorno de desarrollo desplegado sobre contenedores

La solución propuesta para este entorno de desarrollo se basa en contenedores Docker principalmente por algunos de los motivos que se han visto anteriormente: ligereza, no secuestran los recursos del equipo anfitrión, rapidez de ejecución, etc. Al necesitar un solo contenedor por desarrollador, es decir, por máquina, y no ser una aplicación accesible desde el exterior, el hecho de que sean algo más inseguros respecto a las máquinas virtuales no es un factor determinante. De igual manera se ha optado por Docker como tecnología debido a que hoy en día es la tecnología más extendida, madura e implantada, las otras soluciones no han

[4] Del inglés "Command Line Interface" es una método de dar instrucciones a los programas informáticos de manera sencilla mediante una linea de texto simple.

llegado a tener el mercado que tiene Docker, lo cual se traduce en mayor número de recursos disponibles, documentación, soporte, contribuciones, etc.

Existen algunas soluciones comerciales que proporcionan entornos de desarrollo sobre Docker y/o Kubernetes, como por ejemplo Okteto, que ofrece diferentes entornos de desarrollo predefinidos según el lenguaje usado en el proyecto (node, golang, Python, java y ruby). También ofrece la posibilidad de realizar despliegues y pruebas en un clúster Kubernetes alojado en la nube, permite escribir el código usando cualquier IDE y ver los resultados directamente en la nube [41].

Estos entornos proporcionan una distribución estándar con diversas características, el hecho de ser soluciones genéricas y no personalizadas para un equipo de desarrollo concreto hacen que se pierdan algunas de las ventajas como la ligereza o rapidez, dado que estas soluciones incluyen una serie de paquetes preinstalados, los cuales pueden ser excesivos o insuficientes, siendo tediosa la tarea de personalización.

Por otro lado, la imagen aquí diseñada incorpora además una configuración inicial para la utilización de un sistema de control de versiones con las credenciales adecuadas, aspecto que no es tenido en cuenta en las soluciones existentes.

3 TECNOLOGÍAS EMPLEADAS

En el capítulo anterior se han visto de manera global algunos conceptos relativos al control de versiones, contenedores y gestión del ciclo de vida del software. De la misma manera, aparte de esta introducción teórica, se han comparado una serie de herramientas y/o soluciones en cada una de las partes, finalizando cada sección con la elección de una tecnología concreta para cada área.

En el presente capítulo se va a profundizar en cada una de las tecnologías elegidas en las cuales se basa el diseño del entorno de desarrollo, tratando cada una de las tecnologías de manera individual, exponiendo conceptos tanto teóricos como prácticos sobre su origen, funcionamiento, configuración básica y pequeños ejemplos de uso. El capítulo número 4 se centra en la explicación de la implementación concreta.

3.1 GitLab

3.1.1 Historia

Como se ha visto con anterioridad, GitLab es una aplicación web centralizada que integra diferentes herramientas para el despliegue y desarrollo de software, así como para la gestión de proyectos de manera colaborativa y basada en Git como VCS. Una de las razones para elegir GitLab frente a otras soluciones para este proyecto es su versatilidad, puesto que aparte de gestionar repositorios ofrece diferentes servicios como gestión de tareas, seguimiento de errores, wikis, etc.; todo ello bajo una licencia de código abierto.

El desarrollo de GitLab comenzó en el año 2011 de manos de los ucranianos Dmitriy Zaporozhets y Valery Sizov, puesto que ambos querían una herramienta de trabajo para poder colaborar con su equipo de tal manera que no perdieran tiempo en tareas de gestión y pudieran enfocarse en su trabajo diario de manera más eficiente [42].

Esta herramienta fue desarrollada originalmente en Ruby, teniendo en mente el concepto de opensource y su desarrollo estuvo orientado hacia la comunidad. En el año 2012 sus creadores publicaron un post en un importante foro de la comunidad IT donde cientos de personas se inscribieron para usar la versión beta de la aplicación. Un año más tarde grandes compañías usaban GitLab a nivel interno y la demanda de nuevas características era constante, por ello nace GitLab Enterprise Edition, la versión de pago para aquellas compañías que requerían funcionalidades extra [43].

En el año 2015 participan en diferentes procesos junto a aceleradores de startups y mueven su equipo de desarrollo, 9 personas en aquella época, a Silicon Valley. En 2016 hay más de 100.000 organizaciones y millones de usuarios usando Gitlab, lo cual supone a su vez un aumento en el número de personas trabajando para GitLab, también aumentan los beneficios económicos. Desde entonces su uso no ha parado de crecer, siendo una de las principales herramientas para gestionar software, tareas y almacenar código [42].

Ilustración 8 Logo de GitLab

3.1.2 Instalación y configuración

GitLab puede ser instalado de numerosas maneras, la recomendada por los desarrolladores es mediante el paquete GitLab Omnibus el cual está disponible para ciertas distribuciones Linux como pueden ser Ubuntu, Debian, Centos, OpenSUSE e incluso para Raspberry Pi.

Otros métodos de instalación contemplan instalaciones en entornos Kubernetes a través de Helm Charts, Google Kubernetes Engine, Amazon Elastic, RedHat OpenShift; entornos Cloud como Amazon Web Service o Google Cloud Platform; Terraform; Docker; Ansible Playbook y un largo etcétera [44].

Las recomendaciones para instalar GitLab Communty Edition (CE) de manera que satisfaga las necesidades de los usuarios sin ver afectada la experiencia de usuario es la siguiente: Más de 2 cores de CPU y mínimo 8 GB de RAM para soportar 100 usuarios; aunque este proyecto el número de usuarios es significativamente menor [45].

Para instalar GitLab CE en cualquier distribución Linux los pasos requeridos son similares, únicamente cambia el comando del gestor de paquetes, por ejemplo, `yum`, `apt-get`, `dnf`, etc. A grandes rasgos el proceso que se puede encontrar en la documentación oficial es: Instalar algunos paquetes de dependencias necesarias, añadir el repositorio de paquetes de GitLab e instalar el paquete mediante el gestor de paquetes del sistema operativo [44]. En este caso se ha usado el launchpad de Bitnami para su instalación, en el capítulo 4 se profundiza acerca de la instalación específica para este proyecto.

Una vez instalado, accediendo a la URL donde se sirve la aplicación web se crea un usuario y contraseña y ya es posible empezar a usar la aplicación [46].

3.1.3 Conceptos básicos

Todo en GitLab gira alrededor de los proyectos, el usuario puede crear proyectos para alojar el código (repositorio), colaborar revisando el código de otros desarrolladores (merge request), gestionar las tareas relativas a dicho proyecto (incidencias), documentación (wiki) así como implementar acciones relativas a CI/CD como puede ser construir, testear y desplegar el código [11].

Los proyectos pueden ser Privados (el acceso al proyecto debe concederse explícitamente a cada usuario), Internos (el proyecto puede ser accedido por cualquier usuario conectado) o Públicos (el proyecto puede accederse sin ninguna autenticación). Dichos proyectos pueden ser creados desde cero, usando una plantilla o importar proyectos existentes en otras plataformas como GitHub o Bitbucket [47].

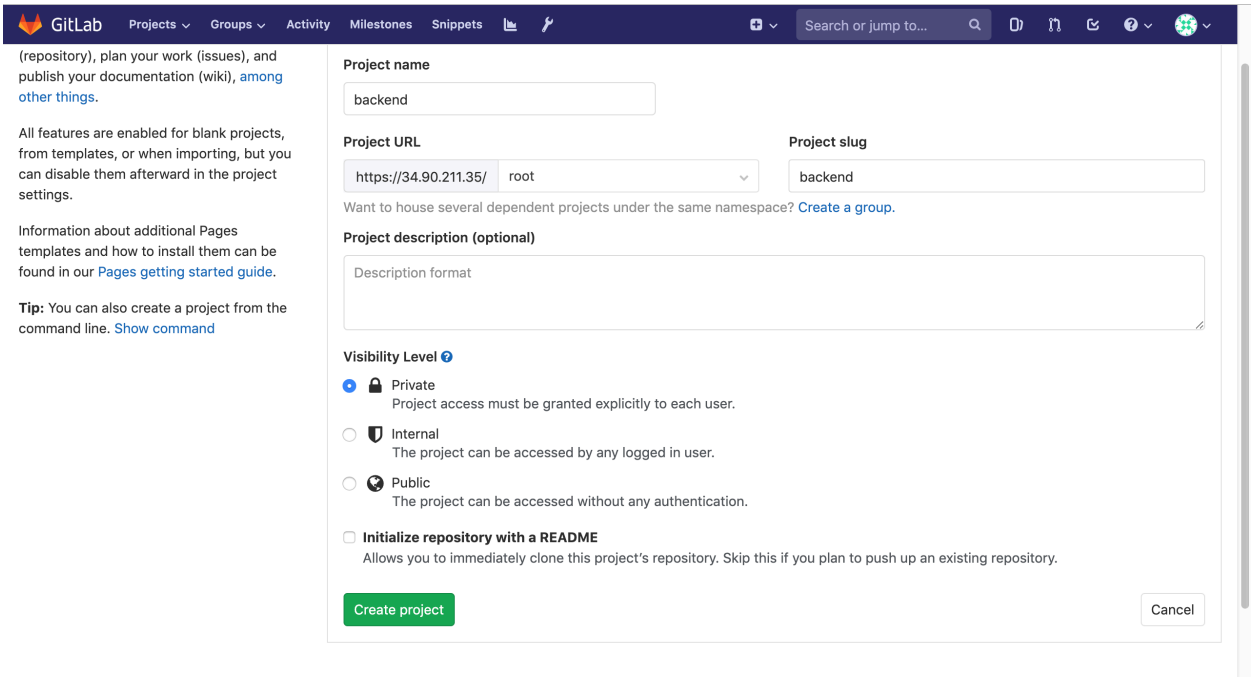

Ilustración 9 Creación de proyectos en GitLab

Una vez creado un proyecto se tiene acceso a diferentes funcionalidades relativas a dicho proyecto. Se parte de una vista inicial en la que se encuentra diferente información relativa al proyecto. Gran cantidad de esta información está relacionada con Git, como el número de commits, de ramas, etiquetas, etc.; también hay información relativa al proyecto propiamente dicho, algunas estadísticas, buscador de archivos, instrucciones para clonar el repositorio, etc. De igual forma, como se verá a continuación, es posible interactuar con los ficheros.

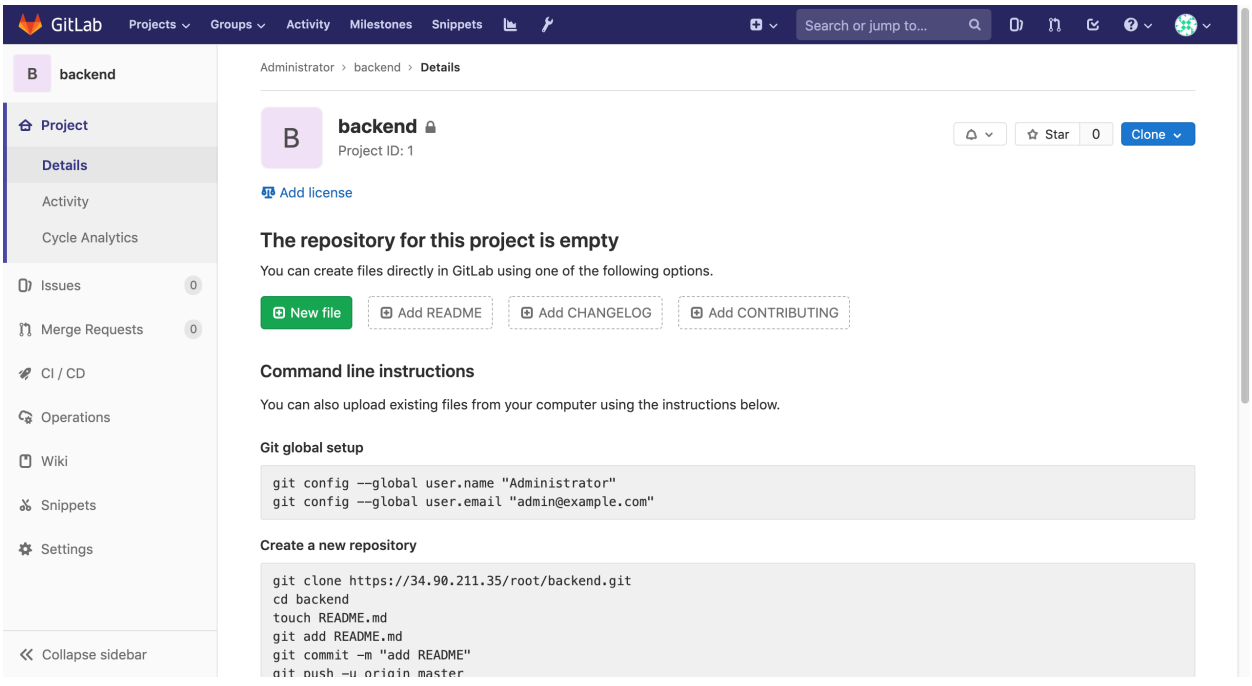

Ilustración 10 Vista principal de un proyecto

Dentro de la vista del proyecto, algunas de las funcionalidades más significativas relativas al proyecto en

cuestión serían las siguientes:

1) **Repositorio** – Almacena el código del proyecto en una plataforma completamente integrada donde se pueden realizar diversas tareas desde un mismo sitio.

   Es posible operar con los ficheros que conforman el repositorio, así como realizar acciones de Git de manera gráfica. Como se verá en el próximo apartado Git permite realizar muchas acciones a través de la terminal; esta sección de GitLab hace que muchas de estas acciones puedan ser realizadas de manera visual.

   Entre las acciones que se pueden llevar a cabo están algunas tareas básicas como añadir un archivo, editarlo (GitLab proporciona un IDE online), hacer commits, ver el histórico del repositorio, etc. De igual forma se pueden ver gráficas y estadísticas relativas al código y a los contribuidores.

2) **Incidencias** – Permite gestionar el proyecto de manera colaborativa con el resto de los miembros del equipo [48].

   Dispone de un tablero para organizar y priorizar el flujo de trabajo, gestionar las tareas de los desarrolladores a través de incidencias. En estas incidencias se pueden añadir fechas de entrega, registrar el tiempo invertido en la tarea, discutir con otros miembros, adjuntar archivos, asociar la incidencia con commits de Git o con Merge requests, etc. [49].

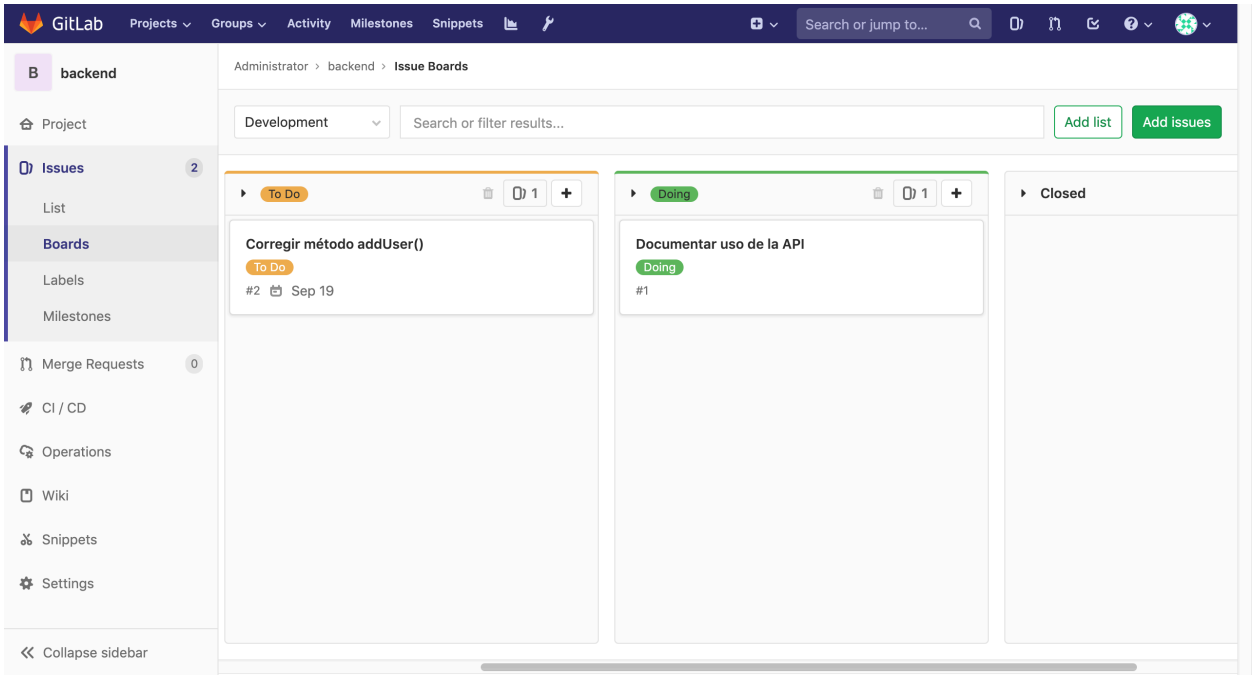

Ilustración 11 Tablero de incidencias

3) **Merge requests** – Aplicando la estrategia de ramas deseada se pueden solicitar revisiones de código a otros desarrolladores.

   De esta manera se puede revisar de forma manual el nuevo código (más allá de los procesos automáticos de CI/CD implementados, en caso de que los haya), todo ello con el fin de mejorar la calidad del código antes de fusionar las ramas con la rama *master* (o aquella otra rama considerada de producción).

   Se profundizará sobre esta característica en el capítulo 4.

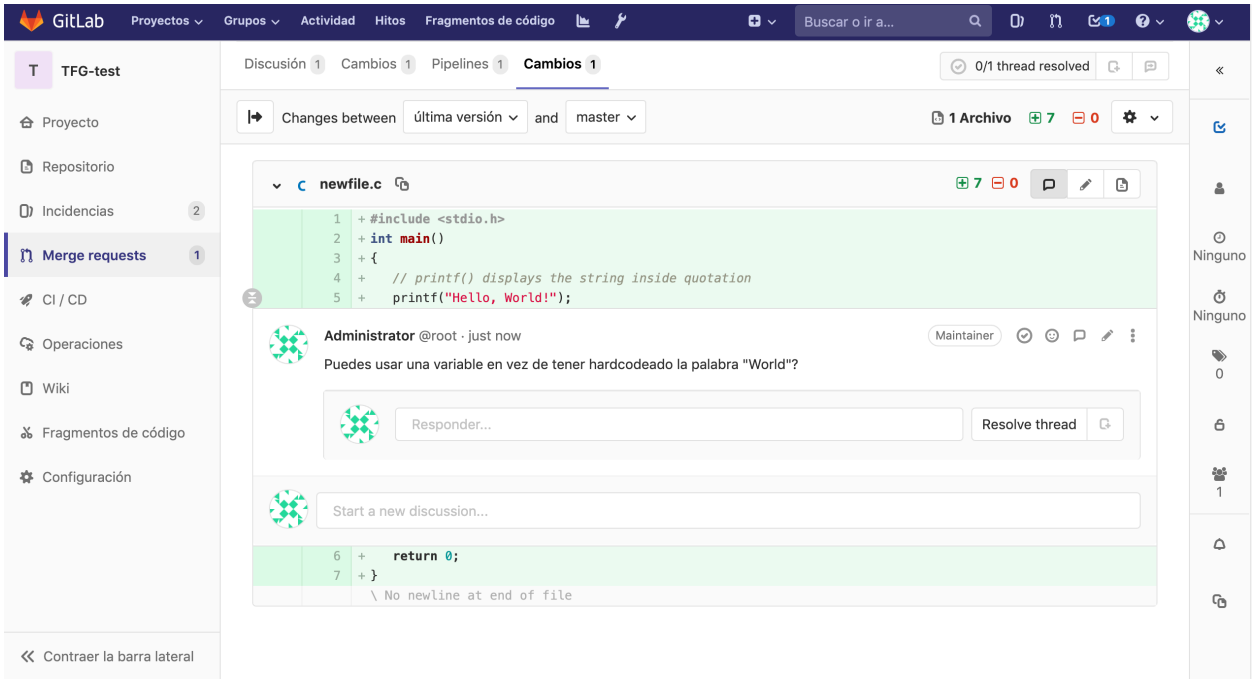

Ilustración 12 Revisión de código

Gracias a este tipo de herramientas no solo se pretende detectar bugs o errores si no también abrir discusiones acerca de la implementación elegida, mejoras en la eficiencia y otros aspectos que son difíciles de cubrir con pruebas automatizadas.

4) **CI/CD** – Herramienta para CI/CD integrada con las acciones de Git y la interfaz de GitLab [3].

   Es una herramienta que viene instalada en GitLab, pero necesita una configuración inicial para adaptarse a las necesidades del proyecto, así como crear los flujos de tareas necesarios y adecuados para cada proyecto o incluso para cada cambio realizado.

   Se pueden configurar las tareas automáticas que se desean ejecutar y qué acción las dispara, por ejemplo, tras cada commit se pueden ejecutar ciertas tareas como comprobar la sintaxis de los cambios introducidos, tests unitarios o si el código sigue las buenas prácticas establecidas; de igual forma para otras acciones de mayor relevancia como puede ser un merge request que pretende fusionar los cambios presentes en una rama con la rama *master*, se pueden lanzar otras acciones, como pueden ser la ejecución de test de integración.

   Toda esta configuración debería ser realizada por el equipo encargado de QA. Este equipo es el encargado de desarrollar una batería de tests tanto unitarios como de integración, así como una política de buenas prácticas y calidad acorde al proyecto. Esto es algo muy ligado al proyecto, tanto por las necesidades del producto como por el lenguaje en el que se desarrolla. Por lo tanto, no es algo trivial que pueda ser implementado de manera genérica para todos los proyectos, es una parte potente que ofrece la aplicación GitLab la cual debe ser configurada por profesionales.

5) **Wiki** – Permite escribir documentación relativa al proyecto.

   Es posible escribir páginas de documentación usando diferentes formatos como markdown y hacer estas páginas navegables, es decir, enlazar unas con otras de manera que resulte cómoda la búsqueda de información y el desplazamiento entre secciones.

Aunque por defecto GitLab trae pre configuradas ciertas estrategias a la hora de organizar el proyecto, como pueden ser las columnas en el tablero de incidencias o la política de asignación de revisores en los merge requests, todo esto es configurable para adaptarse a cada proyecto o a las políticas actuales del equipo de trabajo.

3.2 Git

3.2.1 Historia y puntos fuertes respecto a otros VCS

Como se ha mencionado en el capítulo anterior, Git es un sistema de control de versiones distribuido. Nació entorno al año 2005 de manos de Linus Torvalds, creador del kernel de Linux.

Entre los años 1991 y 2002 el desarrollo y mantenimiento del kernel de Linux, uno de los proyectos de código libre más grande en cuanto a contribuciones de la época, se realizaba a través de parches y ficheros; hasta el año 2002 cuando decidieron empezar a usar un software propietario para el control de versiones, BitKeeper.

En el año 2005 la relación entre la comunidad encargada del desarrollo y mantenimiento del kernel de Linux y los desarrolladores de Bitkeeper se rompió, por lo que la herramienta ya no era ofrecida de manera gratuita a la comunidad, pasando a ser una herramienta de pago. Esto hizo que la comunidad, encabezada por Linus Torvalds, decidiera implementar su propio sistema de control de versiones, así es como nació Git [50].

Ilustración 13 Logo de Git

Tras esta introducción histórica se verán las bases de Git y aquellos aspectos que lo hacen único frente a otros VCS distribuidos.

La diferencia más notable es la forma en la que Git gestiona la información de los datos. Otros sistemas almacenan la versión inicial del fichero para posteriormente ir registrando las modificaciones realizadas sobre los mismos, a modo de diferencias, es decir, guardan una lista de cambios en los archivos a lo largo del tiempo [21].

En cambio, Git almacena el estado completo de los archivos en cada momento en el que ha habido un cambio. Para ser eficiente, si hay archivos que no se han modificado en el momento en el que se guardan nuevos cambios, este archivo no se almacena de nuevo, simplemente se enlaza a la última versión. Esto hace que Git se asimile a un sistema de archivos vitaminado en vez de un sistema de control de versiones tradicional.

Git almacena cada archivo bajo su SHA. Esto significa que si tiene dos archivos con exactamente el mismo contenido en un repositorio (o si cambia el nombre de un archivo), solo se almacena una copia.

Pero esto también significa que cuando modifica una pequeña parte de un archivo y confirma, se almacena otra copia del archivo. La forma en que git resuelve esto es mediante el uso de archivos de comprimidos. Cada cierto tiempo, todos los archivos “sueltos” (en realidad, no solo los archivos, sino también los objetos que contienen metadato y directorios) de un repositorio se comprimen en un archivo de paquete.

El resultado de esto es que un repositorio Git, que contiene los archivos recientes sin comprimir y los archivos antiguos comprimidos, generalmente es dos veces más pequeño que el tamaño de todos los ficheros si no se le aplicara compresión.

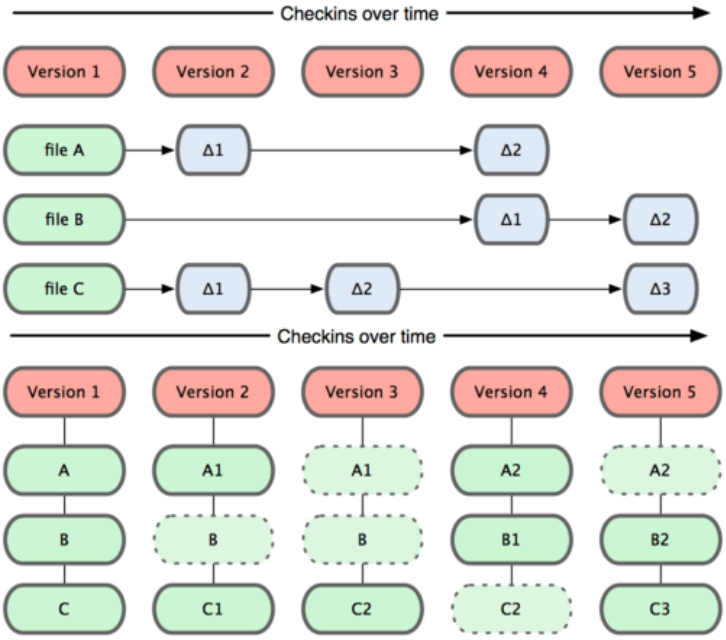

Ilustración 14 Otros VCS (arriba) VS Git (abajo)

Otra ventaja de Git es el hecho de operar localmente sin necesidad de recabar información de otros equipos de la red, incluido el servidor (una vez clonado el repositorio); esto hace que cualquier operación sea más rápida al evitar su paso por la red. Esto se debe a que Git almacena toda la información del repositorio en el disco local, pudiendo consultar toda la historia del proyecto en cuestión sin necesidad de conexión a la red, esta solo es necesaria para subir o traer los últimos cambios realizados por otras personas sobre el repositorio [21].

La mayoría de las acciones realizadas en Git añaden información a la base de datos local (hasta que se suben los cambios al servidor), por lo que es muy difícil perder los datos guardados o realizar alguna acción que no pueda ser deshecha. De la misma manera, Git garantiza la integridad de cualquier acción en cualquier momento puesto que cualquier cambio realizado se verifica de manera unívoca mediante un checksum[5], el cual es usado posteriormente para ver el histórico de cambios, volver a un estado anterior, etc.

3.2.2 Fundamentos

En Git existen tres estados en los que se pueden encontrar los archivos que están siendo controlados:

- Confirmado (*Commited*/*Unmodified*): Los archivos en este estado se encuentran almacenados de manera fiable en la base de datos local. Una vez confirmado pasan a estar sin modificar, es de nuevo el punto de partida, por tanto, este estado es tanto el principio como el final. Todos los cambios surgen y terminan en este estado.
- Modificado (*Modified*): Los archivos han sido modificados pero estas modificaciones no han sido registradas por Git.
- Preparado (*Staged*): Un archivo modificado pasa a preparado para ser incluido en la próxima confirmación (*commit*).

Existen discrepancias sobre si existe o no un cuarto estado. Cuando se crea un nuevo fichero inexistente previamente para Git o se deja de seguir un fichero que estaba siendo controlado por Git, se puede decir que este fichero se encuentra en el estado Sin seguimiento (*Untracked*) [21] [51].

[5] En español, suma de verificación; es una función cuyo objetivo es detectar cambios en una secuencia de datos, protegiendo así la integridad de los datos a los que se le ha aplicado dicha función. Existen diferentes funciones según la complejidad del algoritmo usado, básicamente algoritmos más complejos son más seguros, pero más costosos computacionalmente.

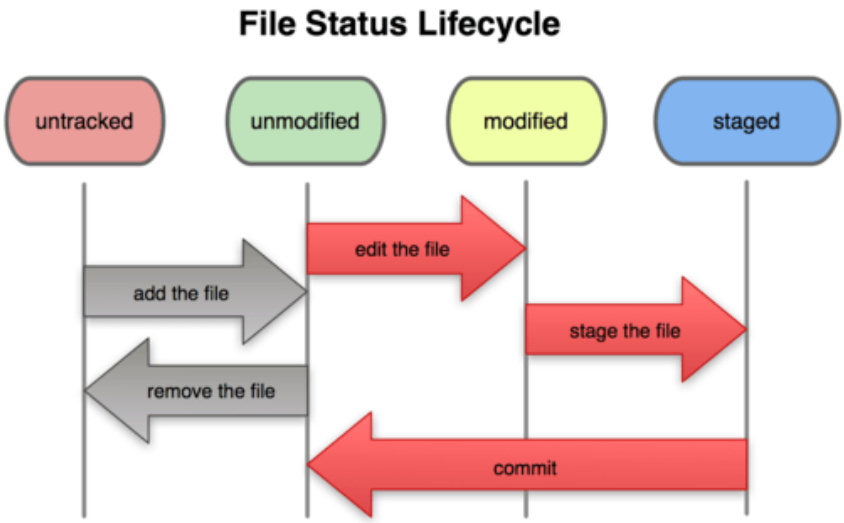

Ilustración 15 Estados de los archivos en Git

Los tres estados de los archivos están íntimamente relacionados con las tres áreas de un proyecto Git:

- Directorio Git (*Git directory*): Es el lugar en el que se encuentra la base de datos con la información del repositorio, también tiene algunos ficheros con metadatos y configuración.
- Directorio de trabajo (*Working directory*): Los archivos presentes en el directorio de trabajo se obtienen a partir de la base de datos presente en el directorio Git. Estos ficheros se descomprimen y se colocan en disco para poder trabajar con ellos.
- Área de preparación (*Staging area*): Técnicamente es un fichero que se encuentra en el directorio Git (anteriormente conocido como índice). Almacena información sobre los cambios que se van a incluir en la base de datos en el siguiente *commit*.

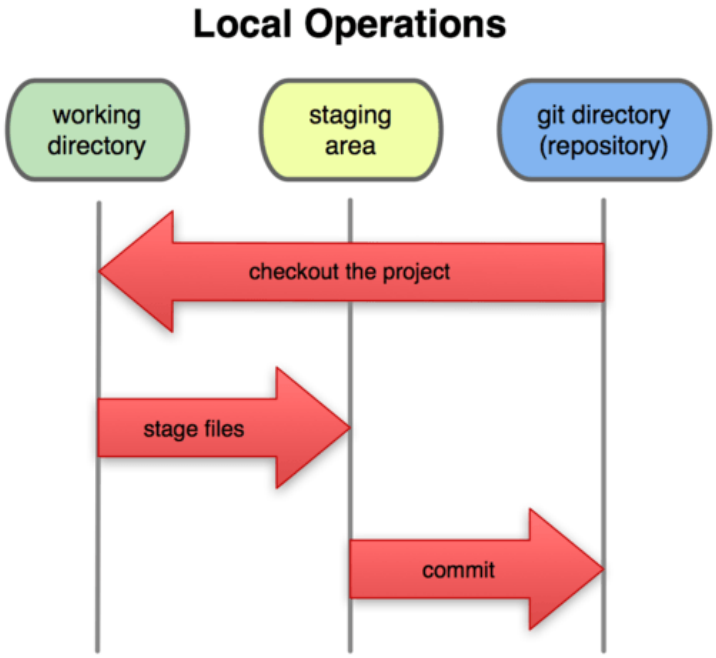

Ilustración 16 Áreas de Git

3.2.3 Instalación y configuración

Git se puede instalar en la mayoría de los sistemas operativos, ya sean Linux, Windows o Mac OS. En este

caso todas las explicaciones serán relativa a Linux, puesto que la imagen Docker usada para el entorno de desarrollo está basada en una distribución Linux [52].

La instalación de Git se puede realizar compilando el código fuente, para lo cual hay que tener instaladas en el sistema algunas dependencias, luego descargar el código fuente, descomprimirlo, compilarlo e instalarlo. Esta forma es más compleja respecto a obtener el binario usando el gestor de paquete incluido en las diferentes distribuciones. La instalación mediante el gestor de paquetesD se muestra en el siguiente bloque de código:

```
$ yum install git-core # Fedora
$ apt-get install git  # Debian
```

Bloque de código 1 Instalar Git

Para comprobar que la instalación ha funcionado correctamente se pueden ejecutar algunos comandos para ver la versión instalada o la ayuda, donde se muestran algunos comandos útiles:

```
$ git --version
git version 2.23.0

$ git --help
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone     Clone a repository into a new directory
   init      Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
   add       Add file contents to the index
   mv        Move or rename a file, a directory, or a symlink
   restore   Restore working tree files
   rm        Remove files from the working tree and from the index

examine the history and state (see also: git help revisions)
   bisect    Use binary search to find the commit that introduced a bug
   diff      Show changes between commits, commit and working tree, etc
   grep      Print lines matching a pattern
   log       Show commit logs
   show      Show various types of objects
   status    Show the working tree status
```

Bloque de código 2 Git en funcionamiento

Una vez instalado Git hay que realizar una configuración inicial para personalizar dicha instalación, esta configuración solo hay que realizarla la primera vez que se instala Git y se mantiene entre actualizaciones. Todo esto respecto a una instalación de Git al uso, en el entorno de desarrollo basado en Docker que se está presentando en este trabajo y como se verá en el siguiente capítulo, esta configuración inicial de Git viene predefinida en la imagen de Docker para que el desarrollador no tenga que realizar estos ajustes.

El binario de Git contiene una herramienta (`git config`) que permite establecer los valores configurables, Git lo hace a través de variables de configuración, las cuales se encargan de configurar el aspecto y funcionamiento de Git. Estas variables se encuentran en tres ficheros diferentes, dependiendo del ámbito al que se deba aplicar esta configuración, de mayor a menor alcance [53]:

- `/etc/gitconfig`: en este fichero se guarda la configuración común para todos los usuarios del sistema y todos los repositorios de cada uno. `git config --system` es el comando para leer y escribir las variables a este nivel.
- `~/.gitconfig`: en este fichero se almacena la configuración relativa a un usuario concreto, en este caso, al usuario cuyo directorio de inicio[6] sea el especificado por Linux. `git config --global` es el comando para leer y escribir las variables a este nivel.
- `.git/config`: hace referencia a un repositorio concreto, el cual contiene el directorio Git como directorio oculto, esta es la opción por defecto si no se le pasa ningún parámetro a `git config`.

Conviene mencionar que la configuración se sobrescribe de la más específica a la más general, es decir, la configuración por repositorio prevalece sobre la de usuario y esta a su vez sobre la del sistema.

El único parámetro de configuración imprescindible es el usuario y la dirección de correo electrónico puesto que estos valores son los que registran de manera inmutable el autor de los cambios en los ficheros. Esta configuración conviene realizarla a nivel de usuario (`--global`) así todos los commits realizados en los diferentes repositorios con los que se trabaje pertenecerán a este usuario. De una forma similar se puede obtener el valor que tiene un parámetro:

```
$ git config --global user.name "Carlos Rodríguez"
$ git config --global user.email carlos@example.com

$ git config user.name
Carlos Rodríguez
```

Bloque de código 3 Configuración de Git

Aparte del usuario y el correo electrónico se pueden configurar una gran cantidad de opciones de todo tipo, desde el editor por defecto hasta el color de los mensajes impresos por la salida estándar, configurar alias para los comandos más habituales y un largo etcétera.

3.2.4 Conceptos básicos

En esta sección se hará un breve repaso de aquellos comandos o flujos de trabajo más habituales para usar Git. Aunque Git es una herramienta muy potente que permite realizar un sinfín de tareas, no es el objetivo de este trabajo servir de guía sobre el uso de Git, aunque al ser un pilar fundamental sobre el que se basa el entorno de

[6] Un directorio de inicio, también llamado directorio de inicio de sesión es el directorio que sirve como repositorio para los archivos, directorios y programas personales de un usuario. Se crea automáticamente un directorio de inicio para cada usuario en el directorio llamado del sistema llamado */home*. El carácter "~" hace referencia a este directorio para el usuario actual del sistema.

desarrollo se realizará un repaso de algunos conceptos básicos sobre el uso de esta herramienta.

Para profundizar al respecto se recomienda visitar la documentación oficial [54].

Inicialización y clonado

Para empezar a trabajar con Git, una vez configurado, lo que hay que hacer es tener un conjunto de ficheros sobre los que realizar el seguimiento. Hay dos formas de obtener un repositorio Git, la primera de ellas es partiendo de un proyecto existente que se quiere controlar mediante Git y la segunda es clonar un repositorio existente en un servidor remoto, ya sea público como GitHub o privado [51] [55] [56].

En el primer caso, partiendo de un proyecto existente hay que situarse en la raíz de dicho proyecto y ejecutar `git init` para crear el directorio Git (./`git`) que se mencionó anteriormente. En este punto se dispone de un repositorio Git, con su directorio Git, directorio de trabajo y área de preparación, pero todavía no hay ningún fichero bajo seguimiento, para empezar a controlar las versiones de los archivos existentes hay que añadirlos al área de preparación y confirmarlos:

```
$ git add *.c
$ git commit –m 'Versión inicial'
```

Bloque de código 4 Añadir y confirmar cambios

El primer comando añade todos los ficheros con extensión "`.c`" al área de preparación mientras que el segundo los confirma, por lo tanto, estos ficheros pasarán a formar parte del directorio Git. La acción *commit* queda registrada mediante el checksum en el histórico de Git, y cualquier modificación sobre cualquiera de los ficheros con extensión "`.c`" que existen en el directorio se verá reflejada a partir de este momento en la historia de Git.

La segunda opción de obtener un repositorio Git funcional es clonar uno existente de un servidor, para ello se hace uso del comando `git clone`, el cual descarga una copia del repositorio con toda la información, inicializando el directorio Git (./`git`) y moviendo al disco la última versión de los ficheros según indica la base de datos, con lo cual se crean los ficheros en el directorio de trabajo, de esta manera el repositorio queda listo para empezar a trabajar en él. Por defecto el directorio se crea usando el nombre del repositorio, aunque al comando `git clone` se le puede especificar otro nombre.

Existen diferentes protocolos para descargar los repositorios: *git://*, *http://*, *https://* o *user@server:/path.git* (SSH). En el siguiente bloque de código se clona el mismo repositorio usando el protocolo *git://*, la diferencia entre ambas instrucciones es que en el primer caso el directorio destino del clon mantiene el nombre del proyecto, es decir, *project*, mientras que en el segundo caso el repositorio es clonado a un nuevo directorio de nombre *ProyectoC*.

```
$ git clone git://github.com/carrodher/project.git
$ git clone git://github.com/carrodher/project.git ProyectoC
```

Bloque de código 5 Clonar repositorio

Flujo de trabajo habitual

Una vez que se dispone, a través de uno de los métodos expuestos anteriormente, de un repositorio Git listo para trabajar en él se va a explicar el flujo de trabajo habitual, sin entrar en detalles de uso avanzado de la herramienta.

La herramienta o comando más importante para ver el estado del repositorio en el momento actual es `git status`, mediante este comando se puede determinar qué archivos están en el directorio de trabajo (modificados) y cuáles están en el área de preparación (preparados), los ficheros confirmados que no tienen modificaciones no aparecen. Esta herramienta también lista aquellos ficheros que no están bajo seguimiento porque han sido añadidos al sistema de ficheros recientemente pero no han sido seguidos por Git [51] [55] [57].

Siguiendo el ejemplo visto anteriormente, se tiene un repositorio Git con algunos ficheros con extensión "`.c`" confirmados, por lo tanto, como es de esperar en primera instancia, el comando `git status` no debería mostrar ningún cambio. Para ver las diferentes fases se va a 1) modificar uno de los ficheros confirmados (file1.*c*) y pasar al área de preparación, 2) modificar un fichero confirmado (*file2.c*) y dejarlo en el directorio de trabajo y 3) añadir un nuevo fichero (*file3.c*) que al principio estará sin seguimiento.

```
$ vim file1.c           # Modificar file1
$ vim file2.c           # Modificar file2
$ touch file3.c         # Crear file3
$ git add file1.c       # Pasar a staging file1
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   file1.c

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   file2.c

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    file3.c
```

Bloque de código 6 Flujo de trabajo y áreas de Git

Si en este punto se realiza la acción de confirmar `git commit -m 'Cambios en file1.c'`, los cambios en el fichero *file1.c* pasarían a la base de datos de Git y quedaría registrado mediante el checksum de esta operación. El resto de los ficheros seguirían en el estado actual.

El comando `git diff` muestra los cambios de aquellos ficheros modificados (los que aún no se han movido al área de preparación), en este caso *file2.c*:

```
$ git diff
diff --git a/file2.c b/file2.c
index 3b7617..479abe5 100644
--- a/file2.c
+++ b/file2.c
@@ -1,6 +1,7 @@
#include <stdio.h>
int main ()
{
-   printf("Hello, World!");
+   char myString[] = "World";
+   printf("Hello, %s!", myString);
    return 0;
}
```

Bloque de código 7 Diferencial de cambios

Este comando muestra de forma diferencial las líneas añadidas, modificadas o eliminadas, de tal manera que se puede ver qué cambios son los que se van a mover al área de preparación.

Una vez revisados los cambios, y si son correctos, se sigue el flujo de trabajo habitual, es decir, mover los ficheros deseados al área de preparación mediante `git add .`, `git add *.c` o `git add` *file2*`.c`, según si se quiere mover todos los ficheros del directorio actual, todos los ficheros con extensión "`.c`" o solo el fichero *file2*.c respectivamente, para luego confirmarlos haciendo uso de `git commit -m 'Texto de la confirmación'`.

Histórico de confirmaciones

Tras haber realizado diferentes modificaciones y sus correspondientes confirmaciones se pueden ver los cambios que se han llevado a cabo haciendo uso de `git log`. Aparte de las confirmaciones, este comando muestra información relativa al autor, fecha, ramas y fusiones:

```
$ git log
commit 50e316e1e30d9c5ae8ddf06d7013607f6bc430ec (HEAD -> master)
Author: Carlos Rodriguez Hernandez <crhernandez@bitnami.com>
Date:   Fri Aug 23 01:18:45 2019 +0200

    Usar variable para printf

commit fb78afba189751d5dbe2ea918c412d7a79494675
Author: Carlos Rodriguez Hernandez <crhernandez@bitnami.com>
Date:   Fri Aug 23 01:10:20 2019 +0200

    Version inicial
```

Bloque de código 8 Revisar histórico de cambios

Como se mencionó al inicio de este capítulo, cada confirmación se encuentra registrada de manera unívoca mediante el checksum (también se puede apreciar el usuario y el email del responsable de la confirmación). Las modificaciones se muestran en orden cronológico inverso, es decir, las más nuevas se muestran al principio. Este comando es realmente útil para navegar por la historia del repositorio, haciendo uso del checksum de cada *commit* se puede volver a ese estado, restaurar los ficheros que había en ese momento, etc [58].

Otro comando útil es `git show`, el cual permite ver los cambios que se hicieron en una confirmación concreta, similar al resultado obtenido por `git diff`, pero relativo a cualquier punto de la historia y no solo a los cambios que existen actualmente en el directorio de trabajo. En el siguiente bloque de código se puede apreciar un resultado muy similar al obtenido anteriormente tras ejecutar `git diff`, en cambio esta vez, este resultado es relativo a un cambio realizado con anterioridad y el cual ya está confirmado, pudiendo de esta manera ver cambios hechos en cualquier punto de la historia.

```
$ git show cf2dffcae007e9105e0a48df2ac64c21abf0770e
commit cf2dffcae007e9105e0a48df2ac64c21abf0770e (HEAD -> master)
Author: Carlos Rodriguez Hernandez <crhernandez@bitnami.com>
Date:   Fri Aug 23 13:44:34 2019 +0200

    Usar variable para printf

diff --git a/b.c b/b.c
index 3b7617c..479abe5 100644
--- a/b.c
+++ b/b.c
@@ -1,6 +1,7 @@
 #include <stdio.h>
 int main()
 {
-   printf("Hello, World!");
+   char myString[] = "World";
+   printf("Hello, %s!", myString);
    return 0;
 }
diff --git a/c.c b/c.c
new file mode 100644
index 0000000..e69de29
```

Bloque de código 9 Información de un commit a partir de su checksum

Repositorios colaborativos

Todo lo visto anteriormente es relativo a un repositorio y una máquina, es decir, el flujo de trabajo seguido hasta ahora es el de un solo desarrollador trabajando en un proyecto; pero el objetivo de este entorno de desarrollo es que sea colaborativo, por lo tanto, los cambios realizados en el repositorio deben estar visibles para todo el equipo [19] [59].

Cuando el proyecto se encuentra en un estado que se desea compartir con el resto del equipo, se tiene que enviar a un repositorio remoto, generalmente el mismo repositorio del que se hizo `git clone` cuando se empezó a trabajar en ello. Para tal efecto existen dos comandos básicos `git pull` y `git push` para,

respectivamente, recibir los cambios confirmados del repositorio remoto al local y para mandar al repositorio remoto los cambios realizados en local.

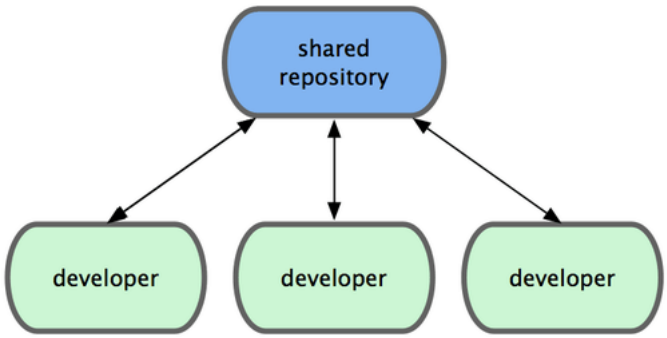

Ilustración 17 Repositorio remoto

El principal problema que surge de esta forma de trabajar colaborativamente son los conflictos, es decir, cuando hay varios desarrolladores modificando los mismos ficheros. En este aspecto pueden encontrarse conflictos cuando intenta poner en común el trabajo llevado a cabo por diferentes personas.

Para Git no hay un desarrollador que tenga la "fuente de certeza" cuando hay conflictos, simplemente el primero que suba los cambios del equipo local al repositorio remoto será el que lo haga limpiamente, el segundo (o siguientes) desarrolladores que hayan hecho cambios en las mismas zonas de los ficheros modificadas por el primer desarrollador deberán resolver estos conflictos de manera manual.

Para ello deberá hacer *pull* de los cambios que actualmente hay en el servidor para aplicar los cambios realizados por él mismo, para ello Git dispone de varias herramientas como `merge` o `rebase` de tal manera que estos conflictos son resueltos mediante la acción del desarrollador, la resolución se hace de manera guiada e interactiva.

Ramas

Conviene recordar el concepto visto al inicio de este capítulo donde se explicó que Git guarda instantáneas del repositorio general, no solo incrementos o diferenciales con las modificaciones realizadas, todo ello usando checksum para distinguir de manera inequívoca cada cambio.

El concepto de rama se puede resumir como un puntero móvil apuntando a una confirmación (checksum) concreta, por defecto la rama *master* se crea con el primer *commit*. Cada rama avanza a la par que las confirmaciones, apuntando siempre a la última realizada [60] [61]. El caso de uso más habitual es usar ramas para desarrollar diferentes funcionalidades a partir de un estado concreto del repositorio, de esta manera se crean varias ramas que inicialmente parten de la misma confirmación para luego divergir conforme se van implementando los cambios.

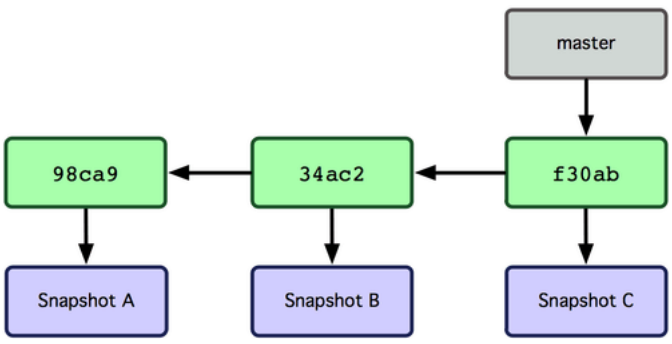

Ilustración 18 Rama *master* e instantánea actual

Git tiene un apuntador especial llamado *HEAD* que diferencia qué rama local está en uso en este momento, puesto que al ser posible crear nuevas ramas e ir haciendo nuevas confirmaciones en cada una de ellas, el estado de las ramas diverge, aunque el inicial fuera una confirmación común. Por defecto, *HEAD* es igual a *master* mientras no haya más ramas creadas en el sistema [62].

Para crear una rama se puede hacer de dos formas, dependiendo si se quiere crear la rama y seguir trabajando en la rama actual (*master* en este caso) o crear la rama y hacer que *HEAD* apunte a la nueva rama para continuar trabajando en esta nueva rama:

```
$ git branch testing        # Crea la rama
$ git checkout -b testing   # Crea la rama y nos sitúa en ella

# Sitúa, a posteriori, en una rama creada mediante git branch
$ git checkout testing      # Nos sitúa en una rama existente
```

Bloque de código 10 Ramas

Mediante los comandos anteriores se ha creado la rama *testing* de dos formas diferentes, la primera de ellas crea la rama, pero mantiene *HEAD* apuntando a *master*. Si en el futuro se desea moverse a la rama *testing* habría que ejecutar el último comando para situar *HEAD* apuntando al mismo commit que *testing*:

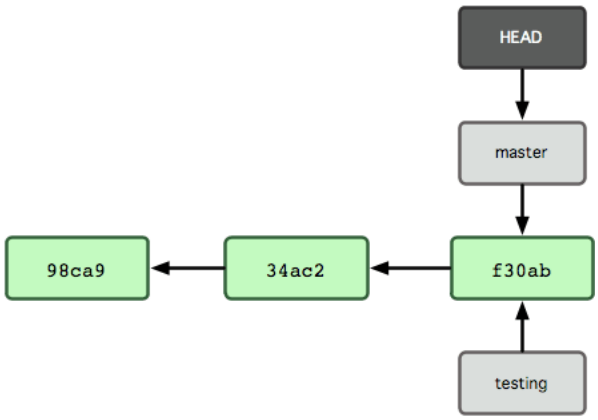

Ilustración 19 Rama *testing* creada desde *master*; *HEAD* apunta a *master* (git branch testing)

Con el segundo comando se crea la rama *testing* y se hace que *HEAD* apunte a ella en un único comando:

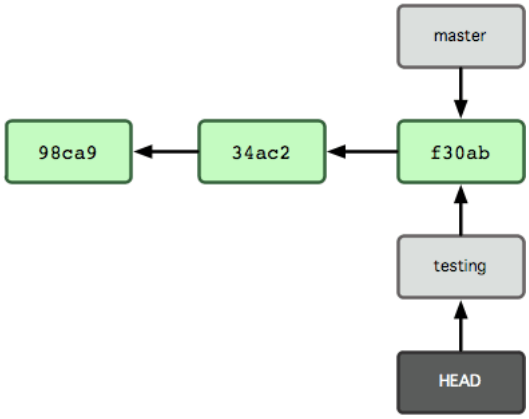

Ilustración 20 Rama *testing* creada desde *master*; *HEAD* apunta a *testing* (git checkout -b testing)

En la siguiente imagen se puede observar este hecho, usando las dos ramas mencionadas anteriormente, se puede ver como hay nuevas confirmaciones en cada una de ellas, de manera que el desarrollo ha divergido.

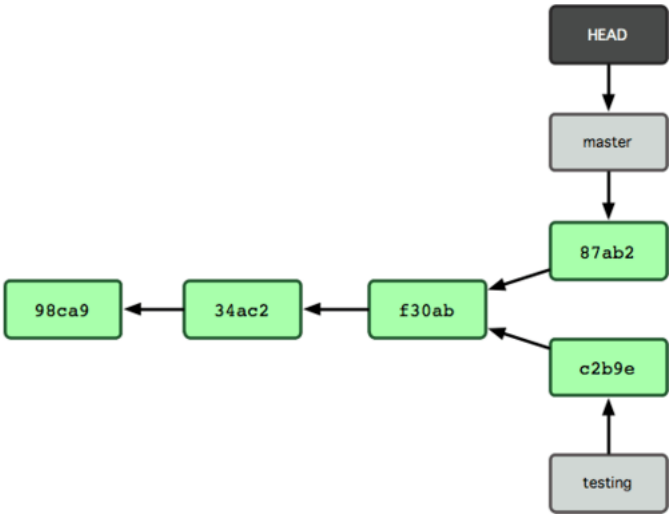

Ilustración 21 Divergencia de *master* y *testing*

En algún momento futuro un desarrollador querrá que dichas ramas se fusionen y el proyecto tenga todas las funcionalidades en las que ha ido trabajando de manera paralela, bien sea para ser subido al repositorio remoto y que el resto de los miembros del equipo tengan las últimas novedades, para ponerlo en producción, para entregarlo al cliente, etc. Las ramas pueden fusionarse unas en otras, aunque haya una gran divergencia desde el punto inicial donde ambas ramas partían de la misma confirmación. Para ello existen diferentes estrategias y métodos para resolver conflictos si los hubiera.

3.3 Docker

3.3.1 Historia

Docker es un proyecto de código libre ideado para automatizar el despliegue de aplicaciones usando contenedores (tecnología de virtualización introducida en el capítulo anterior). Docker nació como un producto interno dentro de la compañía dotCloud, empresa enfocada a ofertas de plataforma como servicio (PaaS), esta compañía fue fundada por Solomon Hykes en el año 2010. Este producto interno, desarrollado sobre proyectos

de código libre, fue liberado en el año 2013, coincidiendo con una nueva versión de Docker que incluía grandes cambios como el reemplazo de LXC (LinuX Containers) por su propia librería (libcontainer) escrita en Go [2] [34] [63] [64].

A finales del mismo año, la empresa dotCloud fue renombrada a Docker, Inc., centrando todo el desarrollo en este proyecto y vendiendo a otra empresa la tecnología y marca anterior. Tras una serie de adquisiciones de pequeñas startups y diversas rondas de financiación, así como acuerdos con grandes empresas del nivel de Microsoft, Red Hat o Amazon; Docker se convierte en la tecnología de contenedores más usada en el mundo, y Docker, Inc. entra en el selecto grupo de "compañías unicornio"[7] [63] [64].

Actualmente Docker Inc. cuenta con diferentes productos en torno a Docker, algunos de ellos orientados al ámbito empresarial (Docker Enterprise Edition (EE)), y por tanto de pago, cuando se hace referencia a Docker en este texto se está refiriendo a Docker Community Edition (Docker CE) .

Ilustración 22 Logo de Docker

3.3.2 Fundamentos

Algunas de las características que han propiciado el auge y la consolidación de Docker como la plataforma de contenedores más usada entre desarrolladores y administradores de sistemas son:

- Flexibilidad: Cualquier aplicación puede ser desplegada sobre contenedores, habrá casos de uso en los que no tendrá sentido desplegar una aplicación usando contenedores, pero no será la tecnología de contenedores en general y Docker en particular la que restrinja su uso, puesto que con las ya mencionadas herramientas de orquestación de contenedores es totalmente seguro el despliegue de estas aplicaciones en producción [2], sin ir más lejos hay numerosas aplicaciones usadas por millones de usuarios desplegadas usando Kubernetes, ejemplos muy sonados son Airbnb o Pokemon Go [30] [65] [66].
- Ligereza: Los contenedores usan y comparten el kernel del equipo anfitrión, por tanto, no se requiere de hardware extra o dedicado para ejecutar contenedores [2] [36].
- Portable: Es posible construir las imágenes Docker localmente, desplegarlas en diferentes repositorios alojados en la nube y ejecutarlas en cualquier sistema operativo [2] [36].
- Dinamismo: Los despliegues basados en contenedores Docker son fácilmente escalables y actualizables, siempre que el despliegue y la configuración se hayan realizado siguiendo unas buenas prácticas, permitiendo al servicio operar sin tiempos de inactividad como consecuencia de tareas de mantenimiento o alta demanda [2].

En los párrafos anteriores, así como en el capítulo anterior se ha hablado de contenedores e imágenes Docker como conceptos genéricos, sin aportar una definición precisa de ambos conceptos. Una imagen Docker es un

[7] Se denomina unicornios a aquellas startups que alcanzan un valor de más de 1.000 millones de dólares. Se eligió el nombre de este animal mítico para destacar la rareza de estos casos exitosos. Se estima que existen alrededor de 280 empresas unicornio a mediados de 2018

paquete ejecutable y autocontenido que incluye todo lo necesario para ejecutar una aplicación concreta: código fuente, librerías, runtime, variables de entorno, ficheros de configuración, etc. Por tanto, un contenedor Docker es una instancia en tiempo de ejecución de dicha imagen, es decir, el resultado de ejecutar una imagen y ocupar un espacio en la memoria del equipo anfitrión [67].

3.3.3 Instalación y configuración

Es posible instalar una versión de escritorio para Mac (macOS) o para Windows (Microsoft Windows 10), de igual manera existen instaladores para la mayoría de los sistemas operativos basados en Linux (CentOS, Debian, Fedora o Ubuntu) y plataformas (*x86_64*/*amd64*, *arm*, *ppc64le*, etc.) [68].

Puesto que la instalación en macOS y Windows se resume en hacer doble clic en el fichero ejecutable (*.dmg* y *.exe* respectivamente) descargado de la web oficial de Docker y seguir el proceso de instalación a través de la interfaz gráfica, se centrará la explicación de esta sección en la instalación de Docker en entornos Linux.

Para entornos Linux hay dos opciones de instalación soportadas de manera oficial: haciendo uso de los repositorios de paquetes o ejecutar el paquete una vez descargado. La segunda opción está orientada a entornos sin conexión a internet, y una de las contrapartidas importantes es el hecho de tener que gestionar manualmente las actualizaciones, teniendo que descargar e instalar el nuevo paquete cuando haya una nueva versión a la que se quiera actualizar [68]. Por tanto, la explicación se desarrollará sobre instalar Docker en entornos Linux mediante el gestor de paquetes.

El primer paso es configurar el repositorio, puesto que los paquetes necesarios no se encuentran en los repositorios por defecto de las distribuciones, para ello hay que actualizar el gestor de paquetes, instalar algunas dependencias necesarias para poder usar repositorios sobre HTTPS, añadir la clave GPG oficial de Docker y añadir el repositorio de Docker al gestor de paquetes del sistema. A continuación, se muestran los comandos correspondientes a las anteriores tareas, ejecutados en el mismo orden en el que han sido enumeradas:

```
$ sudo apt-get update
$ sudo apt-get install apt-transport-https ca-certificates curl gnupg2 software-properties-common
$ curl -fsSL https://download.docker.com/linux/debian/gpg | sudo apt-key add -
$ sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/debian $(lsb_release -cs) stable"
```

Bloque de código 11 Añadir repositorio Docker

Hay que tener en cuenta que los comandos anteriores hacen uso del gestor de paquetes de Debian, para realizar la instalación en otra distribución el proceso es el mismo en cuanto a los pasos necesarios, únicamente es necesario sustituir algunos de los comandos y directrices anteriores, principalmente los comandos relacionados con el gestor de paquetes (`apt-get` y `add-apt-repository`) deben ser sustituidos por sus equivalentes en otras distribuciones (`yum`, `dnf`, etc.) [69] [70].

Una vez que se ha añadido el repositorio oficial de Docker al gestor de paquetes del sistema únicamente hay que actualizar la lista de paquetes disponibles para que se listen los paquetes incluidos en el repositorio de Docker que se acaba de añadir como fuente y luego instalar los paquetes necesarios [68]. En el siguiente bloque de código se muestran los comandos relativos a las dos acciones mencionadas en este párrafo:

```
$ sudo apt-get update
$ sudo apt-get install docker-ce docker-ce-cli containerd.io
```

Bloque de código 12 Instalar Docker desde repositorio

Una vez instalado, para mejorar la experiencia de usuario en sucesivas ejecuciones conviene realizar un pequeño paso tras la instalación para poder ejecutar Docker como usuario no *root*. Por defecto, el demonio Docker se asocia a un socket en vez de a un puerto TCP, este socket es propiedad del usuario *root* y el resto de los usuarios solo pueden acceder a él usando `sudo`, por tanto, el demonio Docker siempre se ejecuta como el usuario *root*. Para evitar anteponer `sudo` en todas las llamadas a Docker es conveniente crear un grupo Unix llamado *docker* y añadir usuarios a él, de esta forma cuando el demonio de Docker se inicia crea un socket accesible para los miembros de este grupo [71]. Las instrucciones se muestran en el siguiente bloque de código:

```
$ sudo groupadd docker
$ sudo usermod -aG docker $USER
$ newgrp docker
```

Bloque de código 13 Añadir usuario docker

Una vez se ha alcanzado este punto, se pueden llevar a cabo otras tareas como configurar que Docker se inicie junto con el sistema haciendo uso de `systemd` o `upstart`, habilitar IPv6, etc [71].

Para comprobar el correcto funcionamiento de Docker basta con ejecutar algunos simples comandos que se muestran en el siguiente bloque de código. El primero de ellos obtiene la versión instalada, mientras que el segundo descarga y ejecuta una imagen cuya aplicación es mostrar un mensaje explicativo por la salida estándar [67].

```
$ docker --version
Docker version 19.03.1, build 74b1e89

$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:451ce787d12369c5df2a32c85e5a03d52cbcef6eb3586dd03075f3034f10adcd
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

Bloque de código 14 Docker en funcionamiento

3.3.4 Conceptos básicos

Partiendo de una instalación funcional de Docker en este apartado se pretende explicar y mostrar a modo de ejemplo algunos conceptos básicos que permitan entender el funcionamiento de la imagen de Docker construida para este proyecto, la cual se desarrollará en el siguiente capítulo.

Como primera aproximación se va a usar como ejemplo el despliegue de una aplicación Python previamente desarrollada, la cual se quiere desplegar en una máquina personal antes de su puesta en producción. Sin usar contenedores, el flujo de trabajo sería algo como: Instalar en la máquina anfitriona todo lo necesario para ejecutar Python, configurar el entorno para que la versión de Python elegida sea la misma que se usará en el servidor de producción y realizar una adaptación del equipo del desarrollador para que el sistema sea lo más parecido posible al servidor de producción donde se desplegará la aplicación.

En el caso de usar una imagen Docker todo esto se resuelve de manera muy sencilla, es tan simple como crear una imagen que contenga la aplicación y use como base una imagen de Python, de esta manera, como se explicó anteriormente, se tendrá una imagen autocontenida y portable de la aplicación que podrá ser ejecutada bajo las mismas condiciones tanto en la máquina del desarrollador como en el servidor de producción, sin tener que configurar nada en el sistema, comparar versiones de librerías, etc.

Dockerfile

Un *Dockerfile* es un fichero que define la imagen de Docker, es decir, todo aquello que va a incluir el contenedor. En esta definición se debe especificar qué ficheros se incluyen, así como los puertos usados puesto

que un contenedor está aislado del resto del sistema anfitrión en términos de interfaces de red y disco, con lo cual hay que definir qué puertos se abren al exterior y qué ficheros de la máquina anfitriona son incluidos en la imagen; imagen que puede ser reutilizada en diferentes equipos una vez construida [72] [73].

Sin entrar a explicar el contenido de la aplicación Python elegida como ejemplo, se puede decir que consta de dos ficheros: *requirements.txt* y *app.py*, el primero de ellos usado para instalar las dependencias necesarias y el segundo es la aplicación propiamente dicha que muestra en el navegador el valor de una variable de entorno levantando un pequeño servidor web. Estos ficheros se encuentran en el mismo directorio que el fichero *Dockerfile*, aunque podría no ser así, solo habría que tener en cuenta la ruta exacta en la definición de la imagen [74].

```
# Use an official Python runtime as a parent image
FROM python:2.7-slim

# Set the working directory to /app
WORKDIR /app

# Copy the application files into the container at /app
COPY app.py requirements.txt /app/

# Install any needed packages specified in requirements.txt
RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME Carlos

# Run app.py when the container launches
CMD ["python", "app.py"]
```

Bloque de código 15 Dockerfile de ejemplo

En este *Dockerfile* se puede ver la definición de una imagen Docker que se llamará *python-example*. Para empezar, se parte de una imagen existente (`FROM`) que contiene Python 2.7, en concreto una versión "slim" que reduce el tamaño de la imagen al no incluir ciertos paquetes y librerías que no son imprescindibles. A partir de esta imagen base se define el directorio de trabajo (`WORKDIR`) y se copian los ficheros de la aplicación (`COPY`), como se dijo anteriormente, el contenedor es autocontenido y portable, por lo que los ficheros necesarios deben ir incluidos en la imagen. Posteriormente se ejecuta (`RUN`) el comando `pip install` para instalar las dependencias, cabe recordar que todas estas acciones se realizan en el contenedor, no en la máquina local. A continuación, se expone el puerto 80 (`EXPOSE`) puesto que es el puerto en el que la aplicación está ejecutando el servicio implementado en el código, exponer significa abrirlo a nivel del contenedor y se define una variable de entorno (`ENV`) de nombre *NAME* y valor *Carlos*. Por último, se define el comando y los argumentos que se ejecutarán cuando se lance el contenedor (`CMD`), el contenedor estará ejecutándose mientras este proceso esté activo, al ser una aplicación de estilo servidor, este proceso permanecerá ejecutándose hasta que sea interrumpido manualmente, en cambio si el comando no fuera un servicio, como podría ser un simple `ls -la`, el contenedor terminaría su ejecución casi instantáneamente, es decir, una vez mostrada la salida del comando `ls -la`.

Construir una imagen Docker

Para construir una imagen hay que ejecutar el comando `docker build` el cual acepta una serie de parámetros, en este caso solo se va a usar la flag `-t` para especificar la etiqueta (*tag*) correspondiente a esta imagen, una etiqueta de la forma "nombre:versión" y la ruta hacia el directorio donde se encuentra el *Dockerfile* [73] [75]. El resultado de este comando es la ejecución de todas las directivas que se encuentran en el *Dockerfile* y la generación de una imagen que se guarda en el repositorio local de imágenes:

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE

$ docker build -t python-example:v1 .

$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
python-example      v1                  03056b6a78a6        5 seconds ago       148MB
python              2.7-slim            36219c2e4e99        9 days ago          137MB
```

Bloque de código 16 Construir imagen

Como se puede ver en el anterior bloque de código, partiendo de un repositorio en el que no se muestra ninguna imagen, tras construir la imagen de esta aplicación se puede apreciar como aparecen dos imágenes, la primera de ellas es la construida en este ejemplo y la segunda es la imagen base utilizada en el *Dockerfile*. Como es de esperar, el tamaño de la imagen *python-example* es mayor que el tamaño de la imagen *python*.

Esta imagen puede ser subida a diferentes registros remotos para ser descargada y ejecutada en otros equipos, para ello hace falta estar registrado en dicho registro mediante `docker login` para poder hacer uso de `docker push` y `docker pull` para subir y descargar las imágenes.

Cada instrucción presente en el *Dockerfile* se traduce en una nueva capa en la imagen generada, es muy importante mencionar esto puesto que reducir el número de capas aporta dos ventajas significativas. La primera de ellas es la reducción del tamaño de la imagen final, y la segunda y más importante, está relacionada con el concepto de caché en el comando `docker build`, al modificar el *Dockerfile* hay que volver a construir la imagen para tener disponibles los cambios realizados; pero Docker implementa una caché que hace que cada capa de la imagen sea guardada en sucesivas construcciones siempre y cuando no haya cambiado la capa en cuestión o algunas de las anteriores.

Esto hace que la construcción de imágenes se vea condicionada por el número de directivas en el *Dockerfile* y su orden. Se intentará poner primero las instrucciones inmutables, es decir, aquellas que no están diseñadas para ser cambiadas con asiduidad, de esta manera se pueden cachear y harán que el tiempo de construcción disminuya.

Ejecutar una imagen Docker

Como se explicó anteriormente, un contenedor es el resultado de la ejecución de una imagen. En los párrafos anteriores se ha construido una imagen que se almacena localmente con el nombre *python-example:v1*, el siguiente paso es ejecutar esta imagen e inspeccionar el contenedor que se está ejecutando mientras el proceso lanzado por el comando presente al final del *Dockerfile* se encuentre en ejecución [72] [75].

Para ello se hace uso del comando `docker run` el cual ejecuta la imagen previamente construida. Para el ejemplo descrito, se usará la flag `-p` para mapear el puerto expuesto por el contenedor (80) con otro puerto en la máquina (4000). Como se muestra en el siguiente bloque de código, en la terminal aparece un mensaje indicando la URL en la cual está accesible la aplicación, pero como este mensaje es generado por el contenedor, el puerto indicado será el 80, puesto que es el que expone el contenedor.

```
$ docker ps

$ docker run -p 4000:80 python-example:v1
 * Serving Flask app "app" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://0.0.0.0:80/ (Press CTRL+C to quit)

$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
       NAMES
49a5a4bbcace        python-example:v1   "python app.py"     About a minute ago   Up About a minute   0.0.0.0:4000-
>80/tcp   exciting_raman
```

Bloque de código 17 Ejecutar imagen

Mediante el comando `docker ps` se pueden ver los contenedores ejecutándose en la máquina en este instante.

Para comprobar que la aplicación que se está ejecutando en el contenedor es completamente funcional, basta con acceder a la URL indicada para comprobar que el servicio es accesible y el resultado es el esperado, se muestra la variable de entorno definida en el *Dockerfile*.

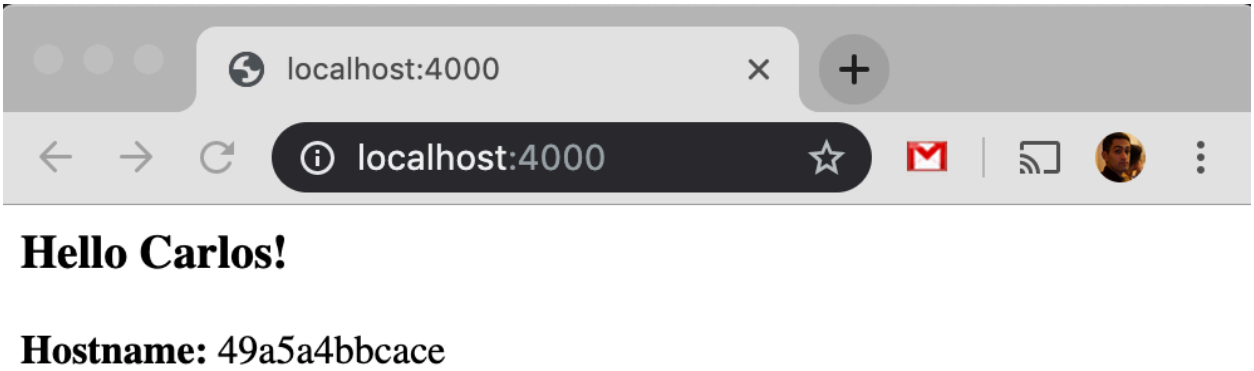

Ilustración 23 Aplicación de ejemplo en funcionamiento

Para mostrar algún ejemplo más acerca del uso de comandos en la llamada a `docker run` se usará una imagen muy ligera llamada *busybox* que contiene algunos de los comandos UNIX más usados en las diferentes distribuciones, esta imagen es muy utilizada para ejecutar pequeñas tareas debido a su ligereza.

Mediante el comando `docker ps -a` se puede listar el histórico de contenedores ejecutados y sus comandos, así se puede observar como pasándole un comando al contenedor, este se ejecuta y luego finaliza su ejecución, en contraposición al ejemplo descrito anteriormente donde el comando, al ser un servicio, se quedaba ejecutando por tiempo indefinido [76].

```
$ docker run busybox echo "Hello world"
Hello world

$ docker run busybox ls -la
total 44
drwxr-xr-x    1 root     root          4096 Aug 24 15:13 .
drwxr-xr-x    1 root     root          4096 Aug 24 15:13 ..
-rwxr-xr-x    1 root     root             0 Aug 24 15:13 .dockerenv
drwxr-xr-x    2 root     root         12288 Jul 16 01:13 bin
drwxr-xr-x    5 root     root           340 Aug 24 15:13 dev
drwxr-xr-x    1 root     root          4096 Aug 24 15:13 etc
drwxr-xr-x    2 nobody   nogroup       4096 Jul 16 01:13 home
dr-xr-xr-x  184 root     root             0 Aug 24 15:13 proc
drwx------    2 root     root          4096 Jul 16 01:13 root
dr-xr-xr-x   13 root     root             0 Aug 24 15:13 sys
drwxrwxrwt    2 root     root          4096 Jul 16 01:13 tmp
drwxr-xr-x    3 root     root          4096 Jul 16 01:13 usr
drwxr-xr-x    4 root     root          4096 Jul 16 01:13 var

$ docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
2a1d9db7bfd4        busybox             "ls -la"                 7 seconds ago       Exited (0) 6 seconds ago
35a7af48b0ef        busybox             "echo 'Hello world'"     33 seconds ago      Exited (0) 33 seconds ago
```

Bloque de código 18 Comandos ejecutados en el contenedor

Por último, se puede ejecutar `docker run` con la flag `-it` para activar una terminal interactiva accediendo al contenedor, de tal manera que sea posible conectarse al contenedor, el cual permanecerá corriendo mientras esta terminal no sea cerrada:

```
$ docker run -it busybox sh
/ $> ls
bin   dev   etc   home  proc  root  sys   tmp   usr   var
/ $> uptime
 05:45:21 up  5:58,  0 users,  load average: 0.00, 0.01, 0.04
```

Bloque de código 19 Terminal interactiva

En este caso, los comandos son ejecutados dentro del contenedor. El mismo resultado se puede obtener para contenedores que ya están en ejecución, para ello se hace uso del comando `docker exec` y del identificador del contenedor obtenido al listar los contenedores en ejecución, de esta forma se puede acceder de manera interactiva a contenedores en ejecución:

```
$ docker ps
CONTAINER ID    IMAGE          COMMAND         CREATED            STATUS             NAMES
33d707c83f7f    busybox        "sh"            8 minutes ago      Up 8 minutes       optimistic_ishizaka

$ docker exec -it 33d707c83f7f sh
/ $> ls
bin   dev   etc   home  proc  root  sys   tmp   usr   var
/ $> whoami
root
/ $>
```

Bloque de código 20 Terminal interactiva. Contenedor en ejecución

Volúmenes

Como se ha mencionado en diferentes ocasiones, una de las ventajas de los contenedores es la volatilidad de los datos, de tal forma que estos se pierden cuando el contenedor se para, pudiéndose así volver al estado inicial fácilmente [66]. Lo que es una ventaja para ciertas aplicaciones no lo es para otras, puesto que hay escenarios en los que se necesita que los datos persistan entre diferentes ejecuciones, como es precisamente el caso del entorno de desarrollo llevado a cabo en este trabajo.

Para ello existen los volúmenes, que dotan de almacenamiento persistente a los contenedores, evitando que los datos se borren cuando se elimine el contenedor. A grandes rasgos un volumen es un directorio que se monta en el contenedor.

Se va a presentar un ejemplo de escritura y lectura cruzada de datos, es decir, se va a crear un directorio llamado */home/local/tfg* en la máquina local y se va a montar en la ruta */home/contenedor/volumen* dentro del contenedor. Una vez creados ambos directorios se van a crear/eliminar ficheros en ambas direcciones, se va a eliminar el contenedor y se va a levantar otro basado incluso en otra imagen (*alpine*) [77] y usando un nuevo punto de montaje dentro del contenedor */home/contenedor2/volumen2*, los datos van a persistir tras todas estas operaciones:

```
$ ls /home/local/tfg    # Equipo anfitrión
requirements.txt

# Se ejecuta el contenedor de manera interactiva y se comprueba que el fichero
# existente en la máquina local existe dentro del contenedor
$ docker run -v /Users/crhernandez/testing/tfg:/home/contenedor/volumen -it busybox sh
/ $> ls /home/contenedor/volumen/
requirements.txt
# Dentro del contenedor se va a crear un nuevo fichero y eliminar el actual
/ $> rm -rf /home/contenedor/volumen/requirements.txt
/ $> touch /home/contenedor/volumen/test.md
# Se termina la ejecución del contenedor
/ $> exit

$ ls /home/local/tfg    # Se comprueban los cambios en el quipo anfitrión
test.md

# Se monta el mismo volumen en otro contenedor con otra imagen y otra ruta destino
$ docker run -v /Users/crhernandez/testing/tfg:/home/contenedor2/volumen2 -it alpine sh
/ &> ls /home/contenedor2/volumen2
test.md
```

Bloque de código 21 Volúmenes

Como se puede apreciar en el bloque de código anterior, la secuencia (explicada mediante comentarios en el propio bloque de código) de comandos ejecutados tanto en la máquina anfitriona como en los dos contenedores que se han lanzado, operan sobre el mismo directorio, estando disponibles los cambios tanto en el equipo anfitrión como en los contenedores.

Esto hace posible que persistan los datos deseados y se mantenga la posibilidad de ejecutar el contenedor partiendo de la imagen inicial en cualquier momento, facilitando así la vuelta al estado inicial fácilmente.

Creación de imágenes intermedias

Otro efecto colateral de la volatilidad de los datos en los contenedores es acerca de los paquetes instalados o configuraciones realizadas. En el punto anterior se ha solucionado la problemática acerca de compartir y persistir archivos, pero no se ha dado una solución a los paquetes instalados o las configuraciones realizadas,

lo cual no aporta mucha flexibilidad a la hora de personalizar una imagen.

Para el caso concreto de este trabajo, surge la necesidad de que cada desarrollador pueda instalar ciertas herramientas o realizar configuraciones que no sería deseable perder cada vez que el contenedor deje de ejecutarse. Para ello se dotará de ciertos script de inicialización que permiten modificar el *Dockerfile* para incluir algunas directivas que permitan cierta personalización al construir la imagen. Sobre esto se hablará en el próximo capítulo. Pero en el caso de que la imagen sea descargada de un registro público o no disponga de esta característica existe una manera de crear una imagen a partir del estado actual de un contenedor.

En el siguiente ejemplo se va a usar *alpine*, otra imagen ligera para exponer una solución a este problema. Dicha imagen no trae instalada la herramienta `curl` por lo tanto si un usuario desea usar esta herramienta en un contenedor ejecutado a partir de esta imagen deberá instalarla manualmente en cada ejecución. El comando `docker commit` permite crear una imagen a partir del estado en el que un contenedor se encuentra actualmente o a partir de un contenedor que no está en ejecución en este momento:

```
# Se ejecuta la imagen alpine de manera interactiva
$ docker run -it alpine sh
  # El comando no existe
  $> curl --version
  sh: curl: not found

  # Se instala curl
  $> apk add curl
  ...
  OK: 7 MiB in 18 packages
  $> curl --version
  curl 7.65.1

  # Termina el contenedor
  $> exit

# Se listan los contenedores ejecutados en el pasado
$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                       NAMES
a03a19077f29        alpine              "sh"                2 minutes ago       Exited (130) 5 seconds ago   morse

# Se crea una nueva imagen a partir del estado anterior
$ docker commit a03a19077f29 alpine-con-curl
sha256:2f7e2f4e5e8075edc0613ab55c4799eab8acbef097e65c98719cb4006d6009bd

# Se ejecuta la nueva imagen de manera interactiva
$ docker run -it alpine-con-curl sh
  # Curl está instalado
  $> curl --version
  curl 7.65.1
```

Bloque de código 22 Confirmar imágenes a partir de estado intermedio

En bloque de código anterior se crea una nueva imagen *alpine-con-curl* a partir de una imagen *alpine* que no lo traía instalado. Para ello se lanza un contenedor basado en *alpine*, se instala `curl` y se sale de este contenedor. A través del histórico de contenedores ejecutados es posible crear una nueva imagen (*alpine-con-curl*), usando el último estado del contenedor, es decir, una imagen *alpine* a la que se le instaló `curl`. Posteriormente es posible lanzar un contenedor basado en la imagen *alpine-con-git* y comprobar que la herramienta ya está instalada en esta nueva imagen.

Todos los comandos mencionados a lo largo de este capítulo soportan gran variedad de opciones, aquí solo se han mencionado algunas de ellas, para obtener el listado completo se recomienda encarecidamente hacer uso de la flag `--help`.

4 DESPLIEGUE DEL ENTORNO DE DESARROLLO

Tras lo expuesto en los capítulos anteriores y dar a conocer las diferentes tecnologías que jugarán un papel fundamental en el desarrollo de este Trabajo, en este capítulo se mostrará el entorno de desarrollo que se ha llevado a cabo, explicando su despliegue y mostrando ejemplos prácticos de su funcionamiento.

En los capítulos anteriores se ha pretendido introducir las piezas fundamentales de las que consta este entorno de desarrollo. Tras hablar acerca de los contenedores, a continuación, se profundizó en la tecnología de contenedores elegida, Docker. El mismo enfoque se ha seguido para las herramientas de gestión de software y GitLab, así como los sistemas de control de versiones y Git. En este capítulo se verá como la conjunción de las anteriormente mencionadas herramientas y tecnologías se unen para formar un entorno de desarrollo que pretende dar servicio a grupos de desarrolladores, y otros roles relacionados con la gestión del ciclo de vida del software.

4.1 Imagen Docker para desarrollo

Como se mencionó en el primer capítulo, cada miembro del equipo dispondrá de un entorno de desarrollo basado en Docker. Este entorno trae ciertas herramientas instaladas y preconfiguradas, así como diferentes las herramientas necesarias para trabajar con lenguajes de programación: Zsh, Git, Java, C, C++, Ruby, Node o Python, entre otras. De esta manera es posible empezar a desarrollar usando este entorno sin tener que realizar modificaciones, siempre y cuando las herramientas incluidas se adapten a las necesidades de los desarrolladores. En concreto este entorno ha sido desarrollado para satisfacer las necesidades de un equipo de trabajo específico. Es importante mencionar que las herramientas incluidas en la imagen por defecto se han elegido en base a unas necesidades determinadas por el equipo de trabajo que lo va a utilizar.

El desarrollador en su equipo donde realice su trabajo tiene que instalar Docker; para ello puede servirse de las instrucciones de instalación mencionadas en el capítulo anterior. Una vez Docker instalado, para empezar a trabajar desde cero con el entorno de desarrollo hay que clonar el repositorio, construir la imagen y ejecutarla o bien traerse la imagen previamente construida desde un registro remoto. Esto es todo lo que se necesita para empezar a desarrollar usando este entorno.

4.1.1 Estructura del entorno de desarrollo

El entorno de desarrollo está contenido en un sencillo repositorio que, a grandes rasgos, consta de un *Dockerfile* con las instrucciones para construir la imagen y una serie de bash scripts que sirven para personalizar y modularizar la imagen de forma que sea más fácil implementar cambios y gestionar las herramientas instaladas, podría hacerse todo en un único *Dockerfile* pero eso haría más complejo el proceso de añadir herramientas, configuraciones, etc.; en definitiva sería peor la experiencia de usuario para los desarrolladores que necesiten ir un paso más allá y personalizar la imagen. El proceso normal de trabajo es que esta imagen sea construida, almacenada y ejecutada a nivel local. No siendo necesario la subida de la imagen a un registro remoto.

De manera puntual, es posible construir la imagen localmente y subirla a un registro remoto para que pueda ser descargada en otro momento y lugar. Este enfoque simplifica el flujo de trabajo en algunos casos concretos, como por ejemplo el caso de desarrolladores que necesitan trabajar con diferentes equipos, ya sea de manera puntual (en caso de necesidad de continuar el trabajo en casa por motivos personales y no disponer de un equipo portátil o sustitución de la máquina por avería), o de manera sistemática (ante la necesidad de trabajar en las instalaciones de un cliente).

Esta flexibilidad se consigue gracias a la distribución de las herramientas de trabajo y del código. La imagen para el entorno de desarrollo puede ser alojada en cualquier registro de imágenes Docker, público o privado, de igual manera, el código se encuentra alojado en el servidor central con GitLab, por tanto, es posible acceder a la imagen y al código desde cualquier máquina que tenga conexión a internet y las credenciales de acceso adecuadas (usuario/contraseña).

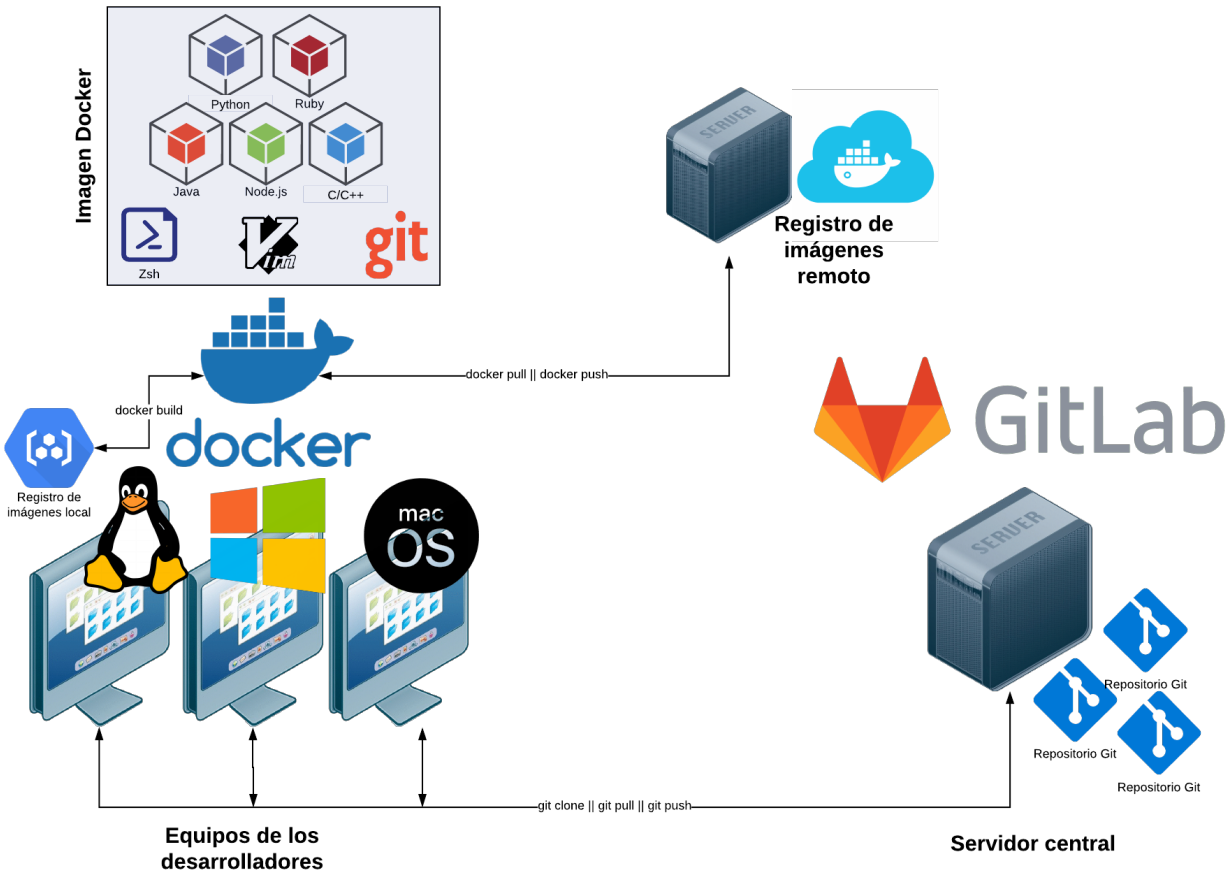

Ilustración 24 Esquema completo del entorno

La estructura del repositorio con los ficheros necesarios para construir la imagen, así como la estructura necesaria para personalizar dicha imagen es la mostrada en la siguiente ilustración [78].

Ilustración 25 Estructura del repositorio

De manera individual consta de los siguientes ficheros y directorios:

- *.gitignore*: Fichero que indica qué ficheros escapan al control de versiones, es decir, qué ficheros o directorios son excluidos por Git. En este caso el directorio *projects* donde se almacena el código de los proyectos en los que trabaja el desarrollador no es necesario incluirlos como parte del control de versiones de la imagen del entorno de desarrollo.
- *Dockerfile*: Fichero con la definición de la imagen.
- *LICENSE*: Fichero de licencia (GPL3).
- *README.md*: Fichero de documentación con instrucciones para construir y ejecutar la imagen, así como explicación de las herramientas instaladas por defecto y cómo personalizar la imagen.
- *execScripts*: Directorio que contiene los diferentes ficheros bash scripts que sirven para personalizar la imagen.
    - *buildScripts*: Directorio donde se almacenan aquellos shell scripts ejecutados en tiempo de construcción de la imagen.
        - *00_configureZsh.sh*: Script que instala y configura Zsh como shell por defecto.
    - *entrypoint.sh*: Script donde se define las instrucciones que se ejecutan en cada ejecución de la imagen.
    - *initScripts*: Directorio donde se almacenan aquellos shell scripts ejecutados en tiempo de inicialización de la imagen.
        - *00_example.sh* y *01_example.sh*: Scripts de ejemplo para inicialización (vacíos).

En la siguiente sección se detalla el contenido del *Dockerfile*, *entrypoint.sh* y 00_*configureZsh.sh*, puesto son los ficheros que determinan las herramientas y configuración incluidas por defecto en la imagen. De igual forma, en el siguiente apartado sobre el flujo de trabajo se explica la manera de personalizar la imagen, por tanto, se desarrolla el contenido de los directorios *buildScripts* e *initScripts*.

Herramientas y configuración por defecto

Para describir las herramientas instaladas por defecto así como la configuración inicial incluida en la imagen

antes de las posibles personalizaciones llevadas a cabo por los desarrolladores o grupos de desarrolladores, se va a hacer un viaje a través de los diferentes ficheros donde se realizan las diferentes instalaciones y configuraciones, de esta manera, la presente sección cumple dos funciones: por un lado listar y describir las herramientas y configuración inicial y por otro lado describir la estructura de los ficheros implicados para facilitar la posterior personalización.

Las herramientas o runtimes instalados por defecto en la imagen son:

- **procps**: Es un paquete que incluye diferentes pequeñas utilidades útiles que brindan información sobre los procesos del sistema. El paquete incluye, entre otros, los programas `ps`, `top` o `free` [79].
- **git**: Como ya se ha mencionado en numerosas ocasiones, esta herramienta de control de versiones es una pieza fundamental en este entorno de desarrollo.
- **vim**: Es un editor de texto altamente configurable para crear y cambiar eficientemente cualquier tipo de texto [80].
- **Zsh**: La shell Z (Zsh) es un shell de Unix que se puede usar como un shell de inicio de sesión interactivo y como un intérprete de comandos para scripts. Zsh es un shell extendida de Bourne Shell (sh) que implementa diversas mejoras, tomadas de otras shells como Bash, ksh o tcsh [81]. De hecho, desde junio de 2019, Apple anunció que la próxima versión de MacOS (Catalina 10.15) adoptará Zsh como shell predeterminada, reemplazando a Bash. En resumen, se puede decir que es una versión vitaminada de sh que engloba diferentes características de otras shells.
- **wget**: Es un paquete de software gratuito para descargar archivos usando HTTP, HTTPS, FTP y FTPS, los protocolos de internet más utilizados [82].
- **unzip**: Es un paquete que sirve para listar, o extraer ficheros y directorios de un archivo ZIP.
- **silversearcher-ag**: Es una herramienta que sirve para buscar cadenas de texto de manera recursiva en los ficheros de un directorio determinado [83].
- **Runtimes**: ***build-essential*** (C\C++), ***python3-pip*** (python), ***default-jdk*** (java), ***ruby-full*** (ruby) y ***nodejs*** (node).

En esta memoria se ha usado Visual Studio Code [84] para la demostración de las características del entorno de desarrollo, aunque en la práctica cada desarrollador puede usar su editor de texto o IDE habitual. Visual Studio Code es es un editor de texto gratuito y de código abierto desarrollado por Microsoft. Incluye diferentes características como soporte para la depuración del código, control integrado de Git, resaltado de sintaxis, finalización inteligente de código, fragmentos y refactorización de código.

Como se ha expuesto al final de la sección anterior, los ficheros involucrados en la instalación y configuración inicial de las herramientas preinstaladas son: *Dockerfile*, *entrypoint.sh* y 00*_configureZsh.sh*.

Dockerfile

```
FROM bitnami/minideb:buster
LABEL maintainer 'Carlos Rodriguez Hernandez <carrodher1179@gmail.com>'

RUN install_packages procps git vim zsh wget unzip silversearcher-ag build-essential python3-pip default-jdk ruby-
full nodejs
COPY execScripts /

# Execute build scripts
RUN for script in /buildScripts/*.sh; do \
        printf '[build] Running %s\n' "$script"; \
        "$script"; \
    done

ENV JAVA_HOME='/usr/lib/jvm/java-8-openjdk-amd64' \
    USER_EMAIL='' \
    USER_NAME=''

WORKDIR /root
ENTRYPOINT [ "/entrypoint.sh" ]
CMD [ "/bin/zsh" ]
```

Bloque de código 23 *Dockerfile* entorno de desarrollo

Dado que el número de instrucciones es pequeño, se va a desglosar el *Dockerfile* instrucción por instrucción para así entender las diferentes directivas que lo conforman.

En primer lugar, aparece la instrucción `FROM bitnami/minideb:buster` [85] que indica la imagen base de la que hereda esta imagen, la cual sirve como punto de partida para las instrucciones siguientes. Como se ha mencionado anteriormente, una imagen de Docker no tiene por qué necesitar un sistema operativo completo, por tanto, surgen diferentes imágenes minimalistas que pretenden incluir los paquetes básicos de un sistema operativo para así aportar ligereza a la imagen.

En este caso se ha usado *bitnami/minideb:buster* [86], una versión ligera de Debian 10 que, aparte de las ventajas en cuanto a tamaño citadas anteriormente, tiene una serie de ventajas adicionales [87]:

- Reducción entorno al 50% del tamaño respecto a la imagen de Docker para Debian 10 (67.5MB frente a 114MB).
- Permite instalar paquetes mediante `install_packages`, un comando que realiza acciones optimizadas del gestor de paquetes por defecto (`apt-get`) como pueden ser actualizar los repositorios o borrar la caché.
- Actualizaciones diarias de los paquetes incluidos por defecto en la imagen base.

La siguiente instrucción `LABEL maintainer 'Carlos Rodriguez Hernandez <carrodher1179@gmail.com>'` [88] no es más que metadato en forma de clave-valor sobre la persona encargada de mantener la imagen, de la misma manera es posible añadir otra información.

La instrucción `RUN install_packages` es una de las piezas fundamentales de esta imagen. En general, la directiva `RUN` hace posible ejecutar cualquier comando en la imagen en tiempo de construcción, lo cual se traduce en una nueva capa de la imagen en la cual se lleva a cabo dicha acción [89].

Es importante mencionar que todos los comandos ejecutados mediante esta instrucción deben de ser compatibles con la imagen base seleccionada puesto que todo se está ejecutando en dicha distribución. No sería posible ejecutar `RUN yum install wget` en una *Dockerfile* que hereda de una imagen base basada en Debian en la cual el gestor de paquetes es `apt-get` en vez de `yum`.

Mediante el comando anterior se instalan los paquetes del sistema y herramientas mencionadas al inicio de esta sección.

Es importante la situación de esta instrucción, como se puede observar es prácticamente la primera instrucción, esto es así por dos motivos: Es probable que los paquetes instalados no se modifiquen con frecuencia, esto no quiere decir que no se personalicen y sean los paquetes por defectos los únicos que se instalen, sino que cuando un desarrollador elige los paquetes que quiere llevar su imagen es algo que no suele cambiar con el

tiempo, por lo tanto se puede considerar una directiva no propensa a cambios. En segundo lugar, instalar paquetes es una de las instrucciones que más tiempo consume. Teniendo en cuenta los dos conceptos anteriores y el sistema de caché mencionado en el apartado acerca de Docker, cabría esperar que una capa que tarda mucho tiempo y sufre pocos cambios se ponga al principio del fichero para que sea cacheada y solo tenga que ser ejecutada una vez, siendo las sucesivas construcciones de la imagen considerablemente más rápidas que la primera.

La siguiente instrucción también es fundamental, `COPY execScripts /` en este caso se están copiando todos los ficheros y directorios que se encuentran en la carpeta local *execScripts* en la carpeta raíz del contenedor. Esta copia se realiza en tiempo de construcción, puesto que es una instrucción del *Dockerfile*, por lo tanto, no se puede pretender modificar estos ficheros y que los cambios aparezcan en una imagen ya construida o en un contenedor en ejecución [90]. Para eso está el volumen que se explicará más adelante.

Esta directiva copia todo el contenido existente en ese directorio de la máquina local a la imagen para que esté disponible en cualquier momento dentro del contenedor, esta es la forma usada para personalizar la imagen; situando en este directorio cualquier script que instale paquetes, configure servicios o realice cualquier otra acción podrá ser ejecutado de manera automática en el contenedor. En los siguientes apartados se profundizará acerca de cómo usar este directorio para personalizar la imagen.

En la siguiente directiva se puede ver la ejecución de un comando mediante `RUN` como se ha explicado con anterioridad, la única relevancia es que el comando es multilínea:

```
# Execute build scripts
RUN for script in /buildScripts/*.sh; do \
        printf '[build] Running %s\n' "$script"; \
        "$script"; \
    done
```

Dicha instrucción ejecuta los scripts situados en la carpeta *buildScripts* en orden alfabético, son por tanto scripts que se copiaron con el anterior comando `COPY` y son ejecutados en tiempo de construcción. En el apartado siguiente se tratan las diferencias entre ejecutar scripts en tiempo de construcción e inicialización.

La siguiente directiva, vuelve a ser una instrucción dividida en tres líneas para facilitar su lectura:

```
ENV JAVA_HOME='/usr/lib/jvm/java-8-openjdk-amd64' \
    USER_EMAIL='' \
    USER_NAME=''
```

La instrucción `ENV` establece variables de entorno que estarán disponible en la imagen [91]. También es posible pasar estas variables de entorno en tiempo de ejecución ejecutando `docker build` con el parámetro `-e VARIABLE=VALOR`.

Mediante la directiva `WORKDIR /root` se establece el directorio de trabajo del contenedor, a partir de este momento todos los comandos son ejecutados relativos a este directorio [92].

Por último, aparecen dos instrucciones muy importantes en cualquier *Dockerfile*, aunque son dos directivas independientes es mejor explicarlas de manera conjunta:

```
ENTRYPOINT [ "/entrypoint.sh" ]
CMD [ "/bin/zsh" ]
```

En primer lugar, la directiva `ENTRYPOINT` permite configurar los comandos que se ejecutarán en el contenedor cada vez que se arranque, a la finalización del proceso lanzado por el último comando, el contenedor muere. Si el último comando ejecutado en el `ENTRYPOINT` es de tipo servicio, el contenedor permanecerá vivo por tiempo indefinido. A su vez, la instrucción `CMD` permite añadir parámetros al contenido de `ENTRYPOINT` [93] [94].

En este caso se ha optado por usar como `ENTRYPOINT` un script y como `CMD` el comando `/bin/zsh`. El script realiza ciertas acciones y por último ejecuta la shell `/bin/zsh` que ha recibido como parámetro. De esta manera hace que el contenedor no muera dado que es un comando interactivo que permanece en ejecución hasta que es matado.

entrypoint.sh

```
#!/bin/bash

if [[ $USER_EMAIL && $USER_NAME ]]; then
  printf '[initialization] Configuring Git with email=%s and name=%s\n' "${USER_EMAIL}" "${USER_NAME}"
  git config --global user.email "${USER_EMAIL}" && git config --global user.name "${USER_NAME}"
else
  printf '[initialization] WARNING! Git not configured\n'
fi

# Execute initialization scripts
for script in /initScripts/*.sh; do
  printf '[initialization] Running %s\n' "$script";
  "$script";
done

exec "$@"
```

Bloque de código 24 Fichero *entrypoint.sh*

Como se ha mencionado en el apartado anterior, la directiva `ENTRYPOINT` determina el comando que se ejecutará en el contenedor y de cuya finalización depende la ejecución del contenedor. Para facilitar la legibilidad del *Dockerfile* y poder ejecutar diferentes acciones en el `ENTRYPOINT`, es posible usar un script para este propósito.

En este caso se usa un script que hace uso de las variables de entorno `USER_EMAIL` y `USER_NAME` que el usuario proporciona al ejecutar el contenedor o al construir la imagen si modifica el Dockerfile, para configurar estos parámetros a nivel de usuario en la configuración de Git. En el siguiente apartado sobre el flujo de trabajo se profundiza en estos aspectos.

Antes de ejecutar el último comando, se ejecutan en orden alfabético todos los scripts de inicialización que el usuario haya situado en el directorio provisto para ello.

Una vez realizada esta configuración inicial, ejecuta el comando que ha recibido como parte de la directiva `CMD`, en este caso ejecuta el binario de la shell Zsh (`/bin/zsh`) el cual se mantiene en ejecución y es el punto de entrada al contenedor. De esta manera cuando el usuario de la imagen acceda al contenedor se va a encontrar una terminal interactiva Zsh y Git correctamente configurado (si las variables de entorno han sido proporcionadas).

00_configureZsh.sh

Como se ha mencionado en diferentes ocasiones en este capítulo, la shell usada para el entorno de desarrollo es Zsh y, por tanto, es este el comando usado para la ejecución del contenedor. En este entorno de desarrollo Zsh se instala como parte de los paquetes del sistema mediante la directiva `RUN install_packages … zsh` del *Dockerfile*.

Aparte de Zsh, también se instala y configura *Oh My Zsh* [95], esto es un framework de código abierto y dirigido por la comunidad para administrar la configuración de Zsh y añadir diferentes funciones útiles, plugins, complementos y temas.

Para configurar Zsh y *Oh My Zsh* se hace uso de un script que descarga, instala y configura este último puesto que no está disponible como paquete del sistema. De igual forma se instalan dos plugins (*zsh-syntax-highlighting* y *zsh-autosuggestions*) y se configura un tema por defecto, sustituyendo el que trae por defecto.

Algunas de las mejoras obtenidas con esta configuración son: autocompletado de comandos mediante tabulador mejorado, corrección gramatical, eficiencia, personalización, expansión de variables, etc.

```
#!/bin/bash

chsh -s "$(command -v zsh)"
wget https://raw.github.com/robbyrussell/oh-my-zsh/master/tools/install.sh -O install.sh
chmod +x install.sh
./install.sh --unattended
rm -rf ./install.sh
git clone https://github.com/zsh-users/zsh-syntax-highlighting.git ~/.oh-my-zsh/custom/plugins/zsh-syntax-
highlighting
git clone https://github.com/zsh-users/zsh-autosuggestions ~/.oh-my-zsh/custom/plugins/zsh-autosuggestions
sed -i 's|ZSH_THEME="robbyrussell"|ZSH_THEME="avit"|g' ~/.zshrc
sed -i 's|plugins=(git)|plugins=(git zsh-autosuggestions zsh-syntax-highlighting)|g' ~/.zshrc
```

Bloque de código 25 Fichero 00_*configureZsh.sh*

Estos temas modifican el prompt[8] de la terminal para mostrar más información como el directorio actual, la rama de Git, el usuario, etc., todo ello de manera visual y colorida, lo cual aporta una mejor experiencia de usuario respecto a las tradicionales terminales. Existe una gran diversidad de temas que pueden ser configurados, tan solo es necesario reemplazar el nombre del tema [96]. En este caso se ha seleccionado *avit*.

Ilustración 26 Tema elegido para Zsh: avit

De igual forma que con los temas, se han configurado dos plugins para mejorar el trabajo de los desarrolladores con el objetivo de facilitar tareas repetitivas, mejorando la productividad. Existen diferentes plugins con diversos objetivos [97]. En este caso se han seleccionado:

- **zsh-autosuggestions**: Es un plugin que sugiere comandos al escribir en la terminal, basa las sugerencias en el histórico de comandos [98].
- **zsh-syntax-highlighting**: Este plugin proporciona resaltado de sintaxis para el Zsh. Permite resaltar comandos mientras se escriben en un terminal interactivo [99].

4.1.2 Flujo de trabajo

En el apartado anterior se han explicado las diferentes partes que conforman el entorno de desarrollo, así como las herramientas y configuraciones que trae por defecto la imagen de Docker. En el capítulo actual se pretenden exponer las diferentes formas de trabajar con dicho entorno dado que es un sistema flexible que permite diferentes formas de ejecutarlo y personalizarlo en función de las necesidades de los desarrolladores y del equipo.

Obtención y ejecución de la imagen

El primer paso es obtener la imagen Docker del entorno de desarrollo, es posible que el equipo de desarrollo

[8] Se llama prompt al carácter o conjunto de caracteres que se muestran en una línea de comandos para indicar que está a la espera de órdenes. Este puede variar dependiendo del intérprete de comandos y suele ser configurable.

construya la imagen y la suba a un registro (público o privado) para que los diferentes desarrolladores se la descarguen o bien existe la posibilidad de que cada desarrollador tenga el repositorio con el código de la imagen y la construya y almacene localmente en vez de en un registro público. También es posible seguir procesos mixtos, es decir, cada desarrollador construye su imagen y además la sube a un registro.

En el caso de que se necesiten realizar personalizaciones sobre la imagen dada por defecto, la mejor opción es que cada desarrollador disponga del repositorio en su equipo y construya la imagen mediante el comando `docker build` cuando se realice algún cambio o personalización en ella.

Aunque cada equipo de desarrollo se puede organizar de una manera diferente según sus propias necesidades y flujo de trabajo actual, en esta sección se va a realizar una propuesta de lo que sería una forma de trabajo distribuida. Para tal efecto se propone alojar el código del entorno de desarrollo de manera que esté accesible para los diferentes miembros del equipo, por ejemplo, GitHub [78].

Cada miembro del equipo puede clonar el repositorio y crear una rama con su nombre para añadir control de versiones a las personalizaciones que vaya realizando a lo largo del tiempo, de esta manera el propio código de la imagen de Docker y las modificaciones que se le realicen quedan registradas en el sistema de control de versiones, Git en este caso. En la siguiente imagen se muestra un ejemplo de tres desarrolladores con sendas ramas de Git en las que cada uno implementa ciertos cambios para personalizar la imagen a su gusto. Cabe mencionar que las ramas de los otros desarrolladores no se ven a nivel local hasta que cada uno no hace push de sus cambios al repositorio.

```
* acb2365 - (origin/master, origin/HEAD, master) Arreglar typo (3 days ago) <Carlos Rodríguez Hernández>
| | * 2a43ab3 - (HEAD -> add) Script para configurar Go (5 days ago) <Maria Donoso>
|/  * aa58dbb - Instalar Go (1 month ago) <Maria Donoso>
| |     * 26ab233 - (HEAD -> removeRuby) Añadir plugin para Zsh (4 minutes ago) <Juan Francisco Azuela Perez>
| |     * 5ef8dcb - Modificar script de inicialización (12 minutes ago) <Juan Francisco Azuela Perez>
|/      * e658db3 - Borrar Ruby, no lo necesito (14 minutes ago) <Juan Francisco Azuela Perez>
* 1cfbd79 - Añadir ejemplos (2 months ago) <Carlos Rodríguez Hernández>
* acb2365 - Initial commit (6 months ago) <Carlos Rodríguez Hernández>
```

Bloque de código 26 Desarrolladores modificando la imagen (vista de Git)

Una vez clonado el repositorio, se accede al directorio donde se ha clonado y se crea una carpeta *projects* que será el directorio donde se almacenen los diferentes proyectos en los que está trabajando el desarrollador. Este directorio será un volumen montado en el contenedor, por tanto, es una carpeta compartida entre el equipo anfitrión y el contenedor.

La carpeta *projects*, ha sido incluida en el fichero *.gitignore* para evitar que añada a la historia de Git los ficheros de los proyectos en los que está trabajando el desarrollador. Los diferentes proyectos incluidos en este directorio tienen su propio sistema de control de versiones y no es parte del código de la imagen, sino datos del trabajo que realiza el usuario.

Es posible que el directorio *projects* se llame de otra manera, para ello habría que crearlo con el nombre deseado, modificar el *.gitignore* y usar la nueva ruta como volumen durante la ejecución del contenedor

```
## Clona el código de la imagen de desarrolo y crea el directorio
## donde se almacenarán los proyectos.
$ git clone https://github.com/carrodher/entorno-de-desarrollo.git
$ cd entorno-de-desarrollo
$ mkdir projects
## Se puede ver que la estructura es la mencionada anteriormente más
## el directorio projects, vacío en este momento pero que contendrá
## los diferentes proyectos en los que trabaje el desarrollador
$ tree
.
├── .gitignore
├── Dockerfile
├── LICENSE
├── README.md
├── projects
└── rootfs
    ├── buildScripts
    │   └── 00_configureZsh.sh
    ├── entrypoint.sh
    └── initScripts
        ├── 00_example.sh
        └── 01_example.sh

4 directories, 8 files
```

Bloque de código 27 Flujo de trabajo 1: clonado del repositorio de la imagen

Una vez que se tiene el código de la imagen hay que construirla y ejecutarla. Para construir una imagen hay que ejecutar el comando `docker build` (Docker debe estar instalado, para ello se pueden seguir las instrucciones mencionadas en el capítulo anterior) en la ubicación donde se encuentra el *Dockerfile*, es decir, siguiendo el flujo de trabajo mencionado anteriormente no es necesario cambiar la ubicación.

Este comando acepta una serie de parámetros, en este caso solo se va a usar la flag `-t` para especificar la etiqueta (*tag*) correspondiente a esta imagen. Una etiqueta es un identificador de la imagen de la forma "usuario/nombre_imagen:versión". El resultado de este comando es la ejecución de todas las directivas que se encuentran en el *Dockerfile* y la generación de una imagen que se guarda en el repositorio local de imágenes.

```
## Se construye la imagen, usando como tag la siguiente nomenclatura:
## usuario/entorno-de-desarrollo:version
$ docker build . -t carrodher/entorno-de-desarrollo:0.0.1
```

Bloque de código 28 Flujo de trabajo 2: Construcción de la imagen

Como se ha explicado a lo largo de los capítulos anteriores, la construcción de imágenes en Docker usa un sistema de caché donde almacena las diferentes capas que conforman un *Dockerfile*, es probable que la primera ejecución de este comando tarde algunos minutos puesto que está instalando todos los paquetes que se

han definido en el *Dockerfile*. En las siguientes ejecuciones la construcción será más rápida siempre y cuando no se modifique la línea de instalación de paquetes que es la que más tiempo consume. Cabe recordar que este comando solo hay que ejecutarlo cuando se realizan cambios en alguno de los ficheros de la imagen (*Dockerfile* o scripts de personalización).

Por último, para ejecutar la imagen y disponer de un contenedor en ejecución, se hace uso del comando `docker run` con una serie de parámetros aparte de la referencia de la imagen construida anteriormente:

- Haciendo uso de la flag `-e` es posible establecer variables de entorno que estarán disponibles en el contenedor, en este caso se usa para configurar el usuario y email usados por Git. Esta flag recibe parámetros de la forma clave-valor.
- La flag `-v` sirve para indicar el volumen, es decir, la correspondencia entre el directorio local y el directorio en el contenedor donde se sincroniza el código de los proyectos, en este caso la carpeta *projects*. La forma de especificarlo es: *directorio-local:directorio-contenedor*.
- Por último, la flag `-it`, la cual no recibe parámetros, para lanzar una shell interactiva y acceder al contenedor.

```
## Se ejecuta la imagen creada anteriormente: carrodher/entorno-de-desarrollo:0.0.1
## -e para establecer dos variables de entorno para configurar el usuario y email de Git
## -v con la ruta local y del contenedor para sincronizar los proyectos
$ docker run \
-e USER_NAME='Carlos Rodriguez' \
-e USER_EMAIL=user@example.com \
-v ~/entorno-de-desarrollo/projects:/root/projects \
-it carrodher/entorno-de-desarrollo:0.0.1
## El comando anterior se ha dividido en diferentes líneas para facilitar su lectura, podría ser en una única línea:
## docker run -e USER_NAME='Carlos Rodriguez' -e USER_EMAIL=user@example.com -v ~/entorno-de-
desarrollo/projects:/root/projects -it carrodher/entorno-de-desarrollo:0.0.1

## Esto es la salida generada por la inicialización del contenedor
[initialization] Configuring Git with email=user@example.com and name=Carlos Rodriguez
[initialization] Running /initScripts/00_example.sh
[initialization] Running /initScripts/01_example.sh

## Este prompt es la shell interactiva del contenedor.
## En este punto el desarrollador está dentro del contenedor, para volver a su máquina
## local basta con ejecutar "exit" o ctrl+D, matando el contenedor
root:~
▶ whoami
root

root:~
▶ exit

## Vuelve a tener el prompt de la máquina local
$
```

Bloque de código 29 Flujo de trabajo 3: Ejecución de la imagen

Ilustración 27 Captura de pantalla real del acceso a la imagen de desarrollo

En este caso los scripts de inicialización no realizan ninguna acción, están vacíos; pero el contenedor detecta los ficheros .sh y los ejecuta mostrando la traza *'[initialization] Running ...'* como se puede ver en la anterior ilustación.

En resumidas cuentas, para empezar a trabajar desde cero con el entorno de desarrollo hay que clonar el repositorio, construir la imagen y ejecutarla. En sucesivas ejecuciones solo habrá que construir la imagen y ejecutarla si se han realizado modificaciones en los scripts de personalización o, lo que será el caso más habitual, solo ejecutar la imagen.

El proceso descrito anteriormente es una de las diversas opciones disponibles, por ejemplo, en el caso de que la imagen esté almacenada en un registro externo, bastará con ejecutar el comando `docker run` usando la tag con la que la imagen haya sido subida.

Editor de texto o IDE

Una de las principales ventajas de este tipo de entornos de desarrollo donde el código está compartido entre el contenedor y la máquina local es el hecho de poder usar el editor de texto favorito que tenga el desarrollador en su equipo, sin necesidad de tener que instalar uno diferente.

Cabe mencionar que para compartir los ficheros no se utiliza ningún protocolo de transferencia o sincronización de ficheros que pueda conllevar retrasos, simplemente se comparte el sistema de ficheros.

En este caso, cualquier repositorio que se encuentre en la carpeta *projects* será accesible tanto desde la máquina local, se suele usar un editor de texto o IDE, como desde el contenedor, usando editores de texto ejecutados en la línea de comandos y sin interfaz gráfica como *Emacs* o *Vim*.

Siguiendo el ejemplo anterior, se van a mover algunos proyectos a la carpeta *projects* del entorno de desarrollo, se va a acceder a dichos proyectos mediante un editor de texto, Visual Studio Code [84] en este caso y se va a mostrar el cambio inmediato tanto en la máquina local como en el contenedor.

```
## Se clonan dos proyectos en los que el desarrollador podría estar trabajando
$ cd projects
$ git clone https://github.com/carrodher/sample-PHP-app.git
Cloning into 'sample-PHP-app'...
remote: Enumerating objects: 6, done.
remote: Total 6 (delta 0), reused 0 (delta 0), pack-reused 6
Unpacking objects: 100% (6/6), done.
$ git clone https://github.com/carrodher/sonarqube-JS.git
Cloning into 'sonarqube-JS'...
remote: Enumerating objects: 85, done.
remote: Total 85 (delta 0), reused 0 (delta 0), pack-reused 85
Unpacking objects: 100% (85/85), done.
```

Bloque de código 30 Flujo de trabajo 4: Situar proyectos en el directorio adecuado

En este caso se han clonado los repositorios con los proyectos de ejemplo, pero estos proyectos podrían estar en la máquina local situados en otro directorio, por lo tanto, solo habría que moverlos a la carpeta *projects* del entorno de desarrollo.

El desarrollador trabajaría en los diferentes proyectos que maneje simultáneamente, podría tener tantos proyectos como deseara, en este momento estos proyectos aparecen tanto en la máquina local como en el contenedor de desarrollo, para ello se va a añadir el directorio *projects* a Visual Studio Code para realizar algunos cambios que se verán reflejados en el contenedor. Una vez realizados los cambios, se puede seguir el flujo de trabajo normal que ha sido descrito en las secciones anteriores, por ejemplo, ejecutar la aplicación, pasar unos tests, subir el código a GitLab para revisión de código, realizar una compilación, etc.

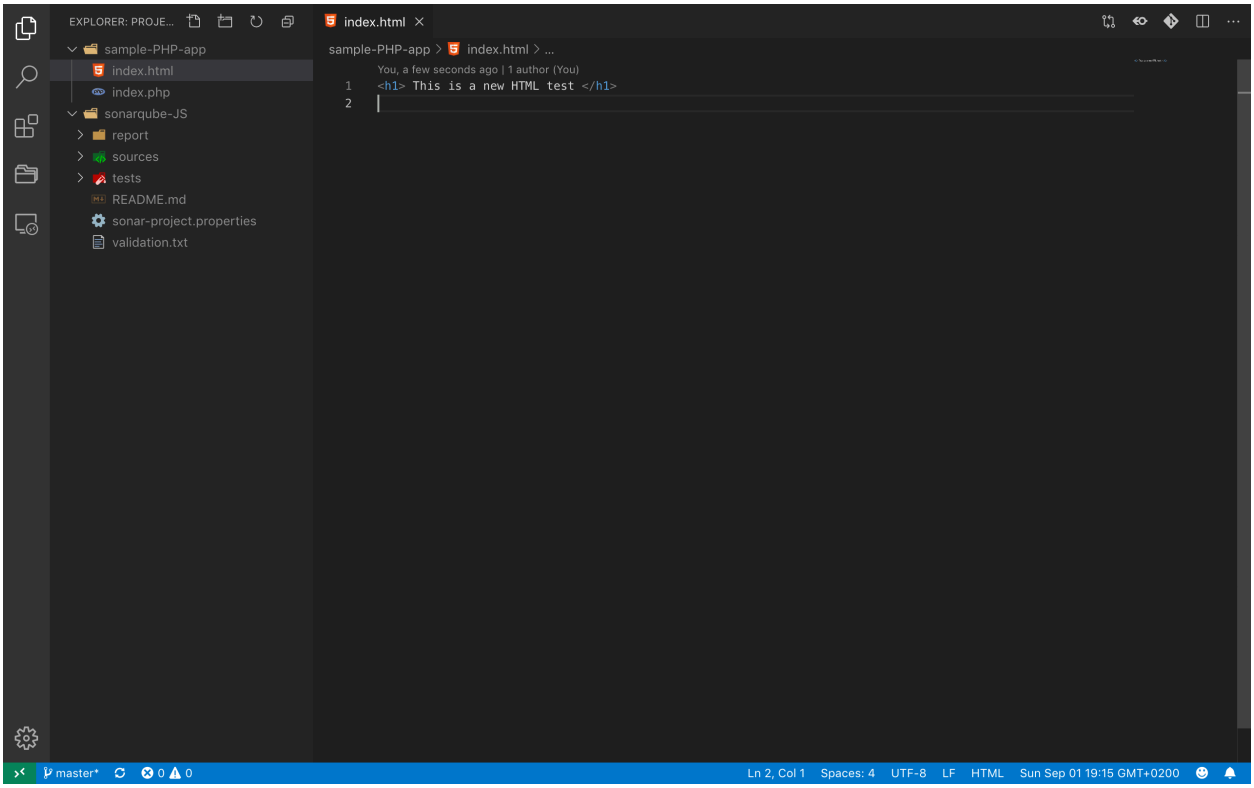

Ilustración 28 Editor de texto en el equipo anfitrión

Como se puede apreciar en la ilustración anterior, se ha abierto el editor de texto con los dos proyectos importados recientemente y se ha realizado un pequeño cambio en un fichero, ese cambio se puede observar de manera inmediata en el contenedor.

```
root:~/projects/sample-PHP-app  master ✗
▶ git diff
diff --git a/index.html b/index.html
index 14f8658..c6eadce 100644
--- a/index.html
+++ b/index.html
@@ -1 +1 @@
-<h1> This is an HTML test </h1>
+<h1> This is a new HTML test </h1>
```

Bloque de código 31 Flujo de trabajo 5: Cambios realizados en local vía IDE aparecen en el contenedor

Personalizaciones

Una vez que se ha explicado el flujo de trabajo habitual para empezar a desarrollar usando el contenedor de este entorno de trabajo se procede a explicar los diferentes métodos disponibles para personalizar la imagen de Docker. Como ya se ha mencionado en diversas ocasiones, la imagen incorpora una serie de herramientas y configuraciones genéricas que pretenden abarcar la mayoría de los casos de usos; pero dado que cada desarrollador tiene unas necesidades específicas es necesario incluir métodos para que esta imagen sea configurable.

Las necesidades específicas del desarrollador pueden venir determinadas por gustos propios, en el sentido de que personalmente se prefiera usar unas herramientas concretas frente a otras, o bien requisitos del proyecto en el que se está trabajando. Sea cual sea el motivo, a continuación, se exponen los diferentes métodos que se proporcionan para realizar estas personalizaciones.

Se van a mostrar 3 métodos para personalizar la imagen, en orden de preferencia o idoneidad, se expondrán razones por las cuales es mejor un método que el siguiente.

Build Scripts e Init Scripts

El entorno de desarrollo se ha diseñado con la posibilidad de crear scripts para realizar personalizaciones; para ello dentro del directorio *execScripts* se pueden encontrar dos subdirectorios (*buildScripts* e *initScripts*) en los que es posible situar estos scripts. El funcionamiento es el mismo en ambos casos, solo difiere el momento en el que los scripts son ejecutados. En ambos casos los scripts incluidos en el directorio en cuestión son ejecutados en orden alfabético, de ahí que sea buena práctica nombrarlos con dos cifras numéricas al inicio para ver claramente el orden en el que se ejecutarán.

Los script situados en el directorio *buildScripts* se ejecutan en tiempo de construcción, es decir, son ejecutados durante la construcción de la imagen mediante el comando `docker build` y se añaden a la imagen Docker como una capa más, por tanto, el resultado de la ejecución del script se incluye por defecto en la imagen.

En este lugar conviene situar acciones que requieran cierto tiempo y no sea necesario ejecutarlas con frecuencia o impliquen configuraciones que requieran otras herramientas instaladas, datos de usuario, etc.

Por defecto se hace uso de un script que se ejecuta en tiempo de construcción para instalar y configurar Zsh y *Oh My Zsh* puesto que es una herramienta que se quiere que vaya incluida por defecto en la imagen.

Por otro lado, están los script situados en el directorio *initScripts* los cuales se ejecutan cada vez que se levanta el contenedor mediante el comando `docker run` y el resultado de esta ejecución se incluye en el contenedor, pero no en la imagen, por lo tanto, deberá ser ejecutado cada vez.

En este lugar conviene situar scripts que realicen acciones personalizadas y que no requieran la descarga o instalación de herramientas para no ralentizar las ejecuciones. Por ejemplo, podría ser útil para realizar configuraciones de Git, del editor de texto incluido en la imagen, puesto que son comandos que no lastran la ejecución del contenedor y son personalizaciones muy concretas.

Zsh y *Oh My Zsh* podrían instalarse en tiempo de ejecución, pero no tiene sentido tener que descargar una herramienta e instalarla cada vez que se ejecute el contenedor, tiene más sentido que esto vaya por defecto en la imagen y solo se instale durante la construcción. Así, la acción de descargar de internet estas herramientas, solo se realiza al construir la imagen, no en cada ejecución.

Tras incluir scripts mediante cualquiera de los dos medios aquí descritos es necesario volver a construir la imagen ejecutando el comando `docker build` puesto que los ficheros para ser ejecutados tienen que copiarse al contenedor, lo cual se hace mediante la directiva `COPY` del *Dockerfile*, aunque los scripts de inicialización se ejecuten durante la ejecución del contenedor.

Modificaciones directamente en el *Dockerfile*

Otra opción es modificar o eliminar el contenido del *Dockerfile*, tras lo cual es necesario volver a construir la imagen.

Un ejemplo de caso de uso es instalar un nuevo paquete en el sistema. Como se mencionó en el apartado anterior, una de las primeras directivas del *Dockerfile* es `RUN install_packages …` que instala una serie de paquetes del sistema. Para añadir un nuevo paquete se puede añadir el paquete en cuestión a la lista de paquetes a instalar especificada en el *Dockerfile*, en lugar de utilizar el método presentado en el punto anterior.

Aunque quizás más útil sea el caso opuesto, como se sabe, en esta imagen se incluyen diversos runtimes para algunos de los lenguajes más comunes como Java, C, C++, Ruby, Node o Python, pero puede darse el caso de que el desarrollador solamente trabaje con Ruby y no necesite ninguno de los restantes lenguajes. Su inclusión en la imagen está aumentando el tiempo de construcción, el tamaño de la imagen, aumentando la propensión a problemas de seguridad, etc. Sería conveniente pensar que para este desarrollador es útil eliminar el resto de los paquetes y solo instalar el relativo a Ruby. Para realizar este tipo de modificaciones solo se puede hacer mediante una modificación directa del *Dockerfile*.

```
diff --git a/Dockerfile b/Dockerfile
index eb32e2c..e67d9c9 100644
--- a/Dockerfile
+++ b/Dockerfile
@@ -1,7 +1,7 @@
 FROM bitnami/minideb:buster
 LABEL maintainer 'Carlos Rodriguez Hernandez <carrodher1179@gmail.com>'

-RUN install_packages procps git vim zsh wget unzip build-essential python3-pip default-jdk ruby-full nodejs
+RUN install_packages procps git vim zsh wget unzip ruby-full
 COPY rootfs /
```

Bloque de código 32 Modificación del Dockerfile para quitar todos los runtimes excepto Ruby

De esta forma, y tras construir la imagen de nuevo, se obtiene otra imagen, la cual se puede identificar mediante una nueva tag, en la que se instalan los paquetes definidos en esta nueva versión.

Como los ficheros que componen la imagen se encuentran bajo el control de versiones, es fácil volver al estado anterior en el que se instalan todos los paquetes. De la misma forma se pueden tener varias imágenes

almacenadas en el registro de imágenes, por tanto, se puede tener una imagen con todos los runtimes y otra solo con Ruby. En el siguiente bloque de código se muestra este proceso, se construye la imagen genérica, sin el *Dockerfile* modificado, a continuación, se edita el *Dockerfile* con los cambios mostrados en el anterior bloque de código, es decir, usando solo Ruby como runtime instalable, a continuación, se construye esta nueva imagen y se ven las dos imágenes disponible. Nótese la diferencia en el tamaño.

```
$ docker build . -t carrodher/entorno-de-desarrollo:completo     # Se construye la imagen sin cambios
$ vim Dockerfile                                                 # Modificar el dockerfile para solo instalar Ruby
$ docker build . -t carrodher/entorno-de-desarrollo:solo-ruby    # Se construye la nueva imagen tras los cambios

# Listar las imágenes en el registro local
$ docker images carrodher/entorno-de-desarrollo
REPOSITORY                          TAG                 IMAGE ID            CREATED             SIZE
carrodher/entorno-de-desarrollo     solo-ruby           40c936eaa7e1        48 seconds ago      256MB
carrodher/entorno-de-desarrollo     completo            a14730747f42        9 minutes ago       1.04GB
```

Bloque de código 33 Nueva imagen a partir de *Dockerfile* modificado

Imágenes intermedias

Este método viene a satisfacer un caso de uso muy concreto, realizar una acción cuyo efecto en principio no persistirá. El programador realiza una configuración o instalación en el contenedor que está usando y desea que persista en nuevas instalaciones de este.

Mediante el comando `docker commit` es posible crear una imagen a partir del estado en el que un contenedor se encuentra actualmente o su último estado en el caso de que ya no esté en ejecución. La contrapartida de este método es que no hay constancia de qué incluye la nueva imagen más allá del nombre elegido en la tag, por lo tanto, no es posible realizar un correcto seguimiento de los cambios introducidos. Aunque es una buena forma de hacer backups del estado del contenedor en momentos puntuales.

```
## Primera ejecución
$ docker run -e USER_NAME='Carlos Rodriguez' -e USER_EMAIL=user@example.com -v ~/entorno-de-
desarrollo/projects:/root/projects -it carrodher/entorno-de-desarrollo:0.0.1

root:~
▶ emacs
zsh: command not found: emacs

root:~
▶ install_packages emacs

root:~
▶ emacs --version
GNU Emacs 26.1

root:~
▶ exit

## Segunda ejecución
$ docker run -e USER_NAME='Carlos Rodriguez' -e USER_EMAIL=user@example.com -v ~/entorno-de-
desarrollo/projects:/root/projects -it carrodher/entorno-de-desarrollo:0.0.1

## Emacs no ha persistido, vuelve a no estar instalado
root:~
▶ emacs
zsh: command not found: emacs

root:~
▶ install_packages emacs

## Se crea una nueva imagen (carrodher/entorno-de-desarrollo:0.0.2-emacs) a partir del contenedor existente
## Con docker ps se obtiene el ID de los contenedores en ejecución
$ docker ps
CONTAINER ID      IMAGE                                  COMMAND                  CREATED             STATUS
fbd7e51526df      carrodher/entorno-de-desarrollo:0.0.1   "/entrypoint.sh /bin…"   4 minutes ago       Up 4 minutes
$ docker commit fbd7e51526df carrodher/entorno-de-desarrollo:0.0.2-emacs

## Se puede ver que ahora hay una nueva imagen que tiene mayor tamaño (por incluir emacs)
$ docker images carrodher/entorno-de-desarrollo
REPOSITORY                          TAG                 IMAGE ID            CREATED              SIZE
carrodher/entorno-de-desarrollo   0.0.1               a14730747f42        2 hours ago          1.04GB
carrodher/entorno-de-desarrollo   0.0.2-emacs         914ab309979b        About a minute ago   1.32GB

## Se ejecuta la nueva imagen
$ docker run -e USER_NAME='Carlos Rodriguez' -e USER_EMAIL=user@example.com -v ~/entorno-de-
desarrollo/projects:/root/projects -it carrodher/entorno-de-desarrollo:0.0.2-emacs

## Emacs instalado por defecto
root:~
▶ emacs --version
GNU Emacs 26.1
```

Bloque de código 34 Flujo de trabajo 6: Personalización mediante imagen intermedia

En los dos primeros métodos, y siguiendo el enfoque propuesto: clonar el repositorio que contiene el entorno de desarrollo y crear una rama con el nombre de usuario del desarrollador; es conveniente ir incluyendo en Git estas modificaciones realizadas en los scripts de personalización o en el *Dockerfile* para así aprovechar las ventajas del control de versiones.

4.2 Herramienta centralizada: GitLab

Tras exponer en el apartado anterior la configuración y funcionamiento del entorno de desarrollo basado en la imagen de Docker, así como el flujo de trabajo inicial que seguiría un desarrollador que usara esta solución, es el turno de explicar el funcionamiento de la herramienta centralizada que alojará los diferentes repositorios de código y servirá para gestionar el proceso software, GitLab.

En el apartado 3.1.2 se mencionaron diferentes formas de instalar GitLab, al ser una aplicación web, el resultado funcional de la aplicación para los usuarios es el mismo independientemente del modo de instalación elegido. La elección del método de instalación, por tanto, debe estar basada en otros aspectos, tales como la infraestructura disponible por el equipo de desarrollo u organización a la que pertenece, presupuesto del que se dispone, requisitos legales, etc.

Para esta demostración se ha instalado la última versión disponible de GitLab (12.2.5) en una instancia de Google Cloud Platform (GCP) a través del launchpad de Bitnami. Concretamente se ha usado una instancia N1-HIGHMEM-2 con 2 vCPUs, 13GB de RAM y con 20 GB de SSD, la cual tiene abierto los puertos 80 y 443 para tráfico HTTP y HTTPS respectivamente, también el 22 para conexiones SSH.

El launchpad de Bitnami [100] proporciona una manera sencilla de instalar diferentes aplicaciones en diversos proveedores de cloud, en este caso se ha elegido GCP como infraestructura debido a la familiarización con su entorno. En una instalación para un entorno real habría que tener en cuenta otros factores, principalmente económicos.

Para realizar dicha se ha partido de una cuenta de Bitnami existente, en caso de necesitar una nueva, se puede encontrar una guía detallada en la documentación oficial. A grandes rasgos, el proceso de creación de una cuenta partiendo completamente desde cero sería: Registro en Google Cloud Platform (60 días y 300$ gratuitos para desplegar recursos) [101], habilitar la API de Google Compute Engine (GCE) [102], registro en Bitnami (gratuito) [103] y por último conectar ambas cuentas para que las aplicaciones lanzadas a través de Bitnami se desplieguen en la cuenta de GCP [104].

Una vez realizado el proceso de registro, para desplegar GitLab en GCP mediante el launchpad de Bitnami el primer paso es acceder al propio launchpad mediante su dirección web [105] y buscar la aplicación en el catálogo:

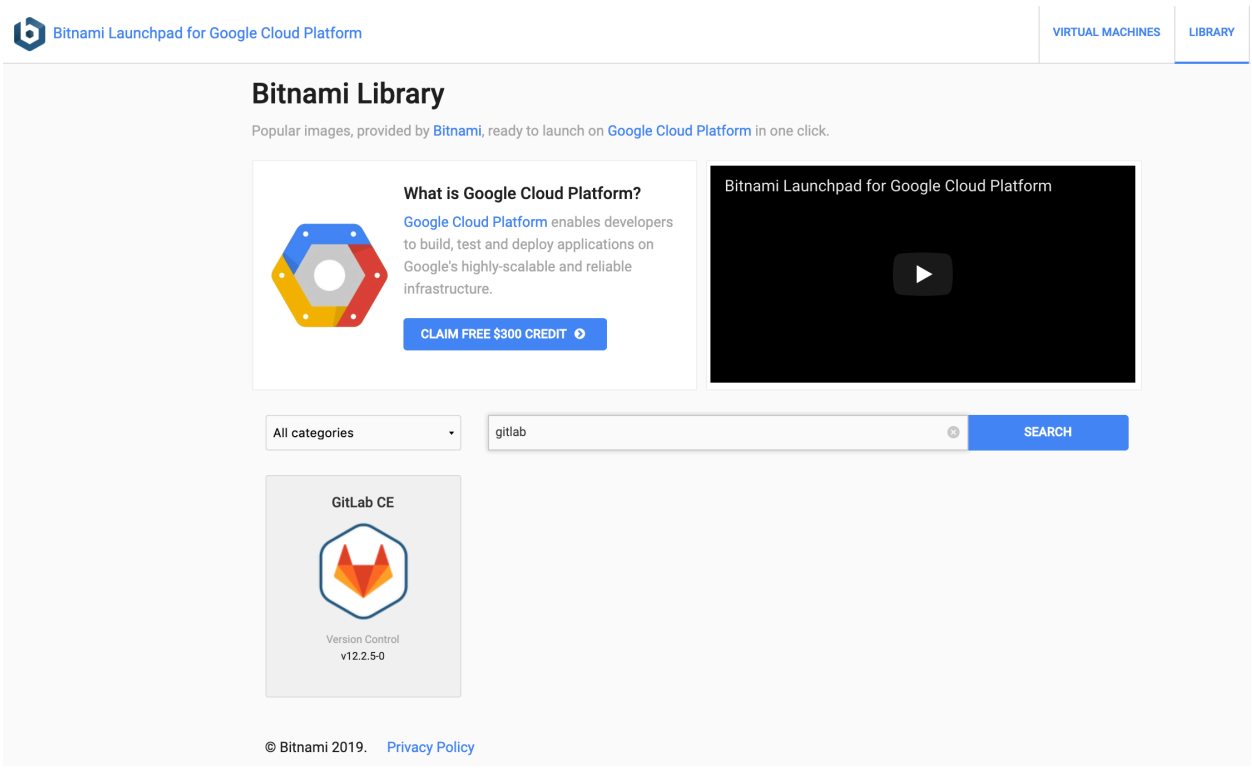

Ilustración 29 Catálogo de Bitnami con la búsqueda de GitLab activa

Haciendo click en el icono de GitLab, aparece una nueva pantalla en la que se pueden configurar diversos parámetros de la instancia que se va a lanzar. Estos parámetros son la región (en este caso se ha dejado la que aparece por defecto), el tipo de disco (solido o magnético), la capacidad del disco, el número de CPUs y tamaño de la memoria RAM entre otras cosas. Tras seleccionar los valores deseados en base al uso que tendrá la instancia aparece una estimación del coste mensual que tendrá esta infraestructura. En este caso alrededor de 70$ mensuales. Este coste se paga íntegramente por los servicios de infraestructura que GCP provee, el uso de Bitnami es gratuito para el usuario, por lo que si se dispone de una cuenta de prueba o cualquier otro descuento proporcionado por GCP el precio estimado que aparece en esta pantalla no aplica.

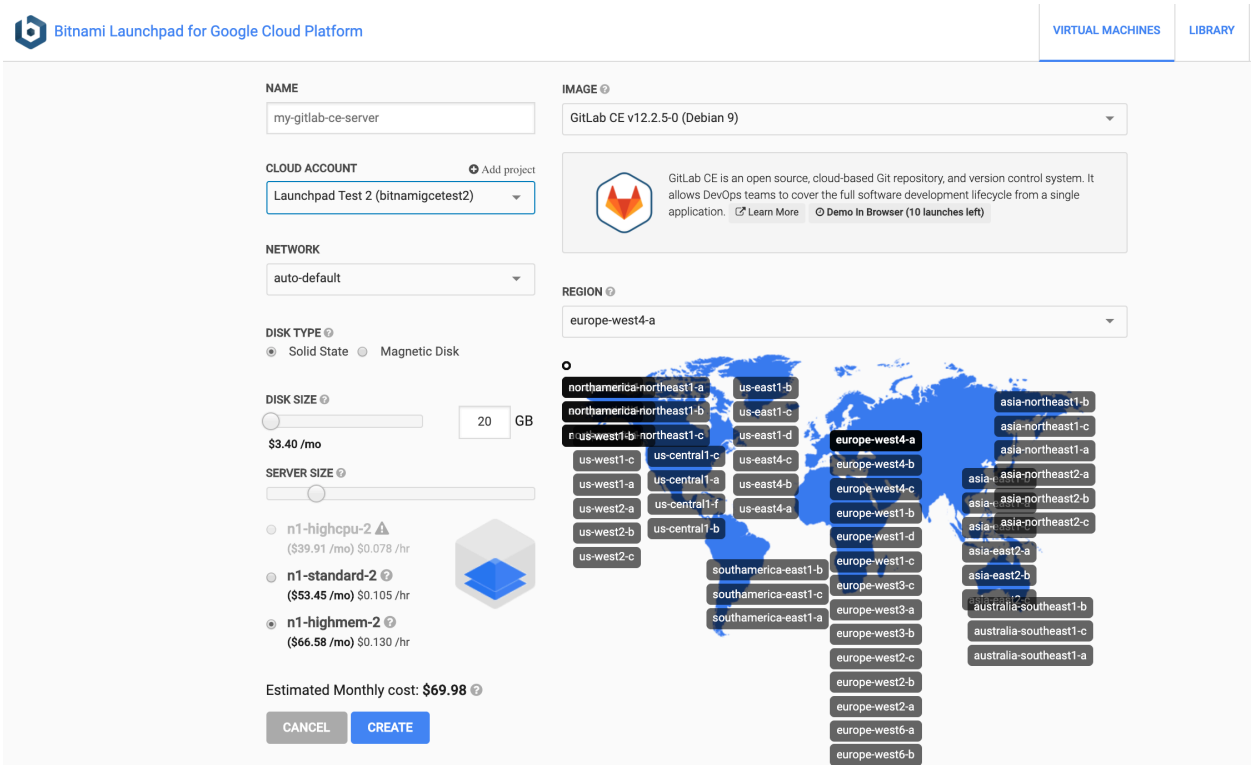

Ilustración 30 Configuración de la instancia

Tras comprobar que los parámetros son los adecuados y hacer click en “Create”, se despliega la instancia en GCP. Desde la propia interfaz de Bitnami launchpad se pueden obtener los parámetros más significativos de la instancia, también aparece un botón “Manage in the Google Console” para acceder directamente a la consola de GCP donde se puede gestionar la instancia de manera directa en GCP, esta pantalla es solo una interfaz intuitiva para facilitar la experiencia de usuario.

En esta pantalla aparece información relevante como el nombre de usuario y contraseña del usuario administrador de GitLab, así como los puertos abiertos y la dirección IP de la instancia. Todos estos parámetros se usarán en las siguientes secciones para acceder e interactuar con la instancia, por ejemplo, la dirección IP será usada tanto para acceder a la interfaz web como para clonar los repositorios.

Por último, desde esta vista también es posible conectarse mediante SSH a la instancia mediante un terminal online, solo en caso de que sea necesario acceder a dicha máquina para realizar tareas avanzadas de gestión o mantenimiento.

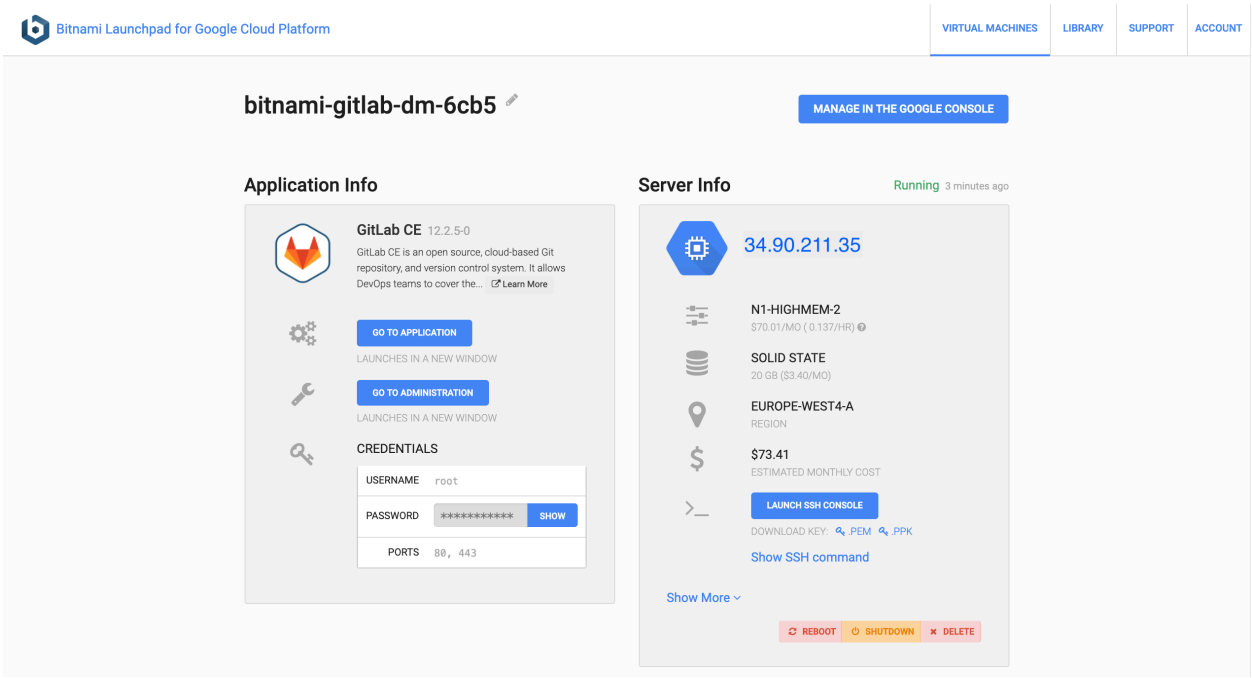

Ilustración 31 Panel de gestión de la instancia desplegada

Como se ha comentado, esta instalación por defecto es provista con un usuario administrador y una contraseña aleatoria para acceder a la aplicación. Puede haber tantos usuarios administradores como se quieran, pero la configuración inicial debe ser realizada por el administrador por defecto que se incluye durante la instalación hasta que, al menos, se cree otro con los mismos permisos.

Accediendo a la dirección IP de la instancia se puede realizar el login, tras el cual se accede al panel principal de GitLab, al ser el primer acceso no aparece contenido en él:

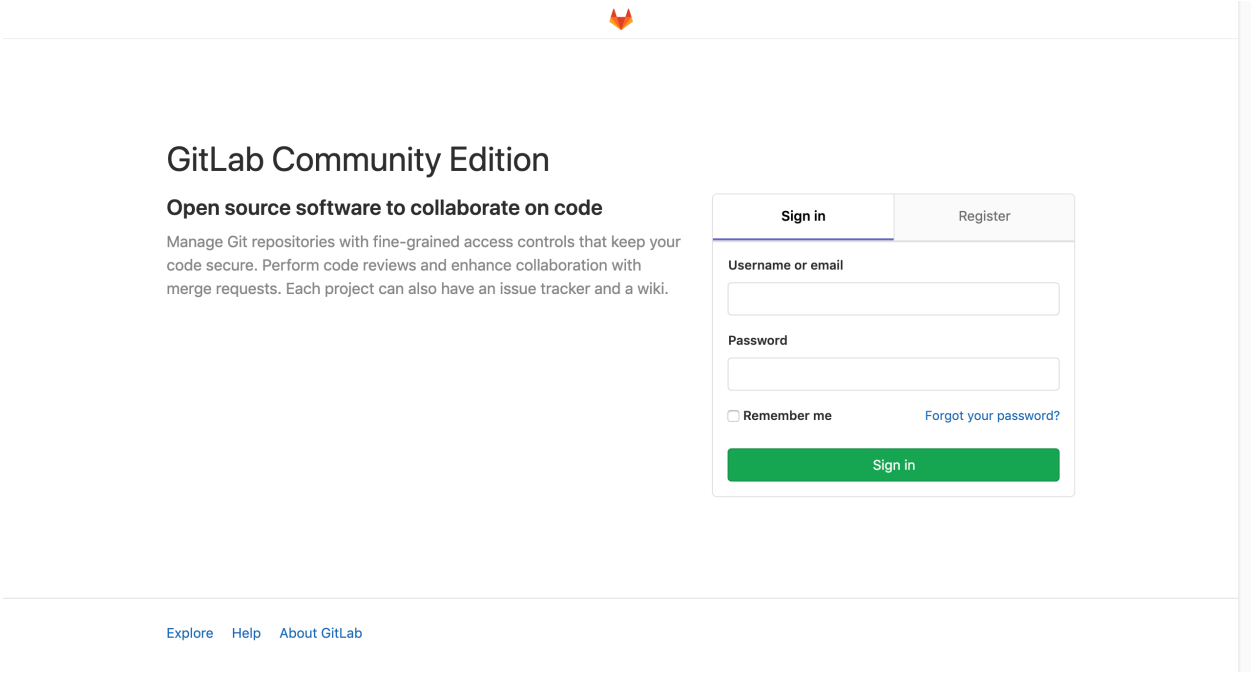

Ilustración 32 GitLab login

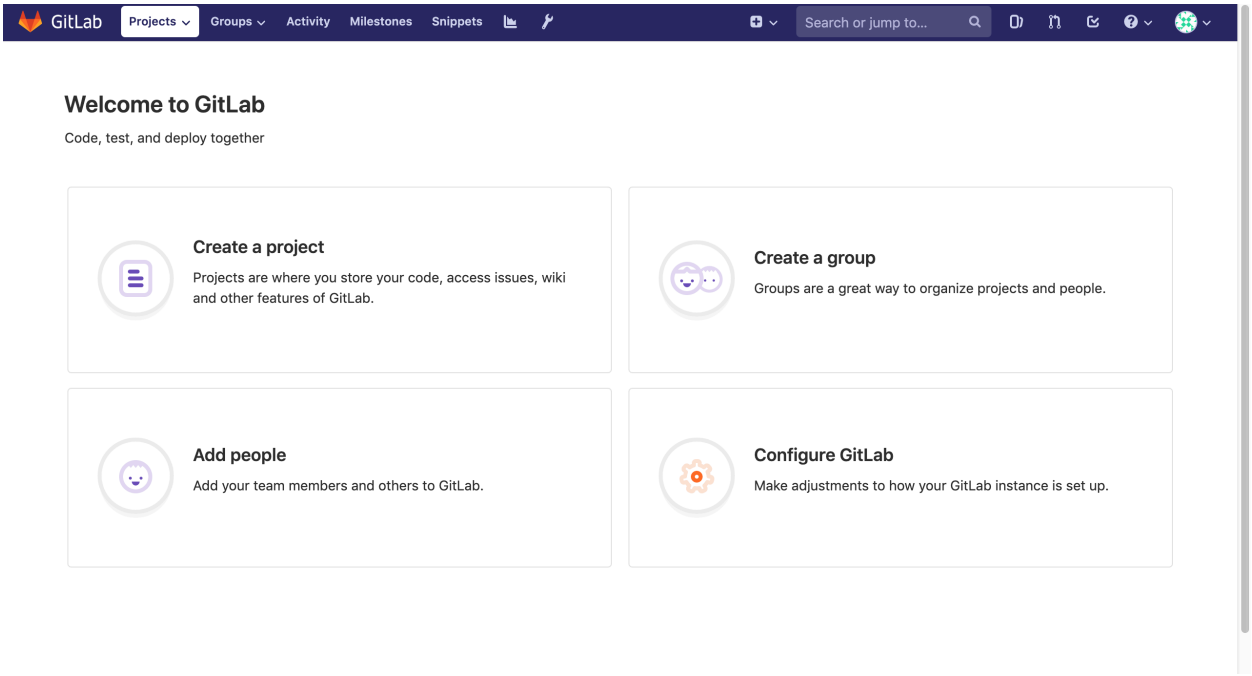

Ilustración 33 Vista principal GitLab

Una vez que se ha comprobado que la instalación ha sido realizada con éxito, el administrador puede realizar una serie de tareas para configurar la herramienta y dar acceso al resto de usuarios, los cuales pueden ser también administradores. La asignación de roles en GitLab dependerá del tamaño del equipo de trabajo, así como de su política de seguridad.

Al igual que la imagen Docker, GitLab es configurable pero no tanto a nivel de desarrollador o usuario sino a nivel de equipo. Hay que hacer énfasis en el ámbito en el que actúa cada herramienta, mientras que el entorno de desarrollo es una herramienta local usada por cada desarrollador y que puede ser adaptada a sus necesidades personales; GitLab es una herramienta centralizada de confluencia para todo el equipo, por tanto, estas personalizaciones se hacen a nivel de todo el equipo.

Obviamente, hay algunos ajustes que se hacen a nivel de cada usuario como puede ser seleccionar la foto de perfil, email, estado, notificaciones, temas, etc. El hecho de que haya parte de la configuración que es específica de cada equipo de trabajo, hace que la persona que administre GitLab necesite tener idea de las necesidades del equipo de trabajo.

En el siguiente subapartado se verán algunas de las configuraciones básicas que son convenientes realizar para que un equipo de desarrolladores empiece a ser productivo con esta herramienta. En el último subapartado se va a exponer cómo sería el flujo de trabajo habitual de un desarrollador (u otro rol como el gestor del proyecto), utilizando para ello los repositorios o proyectos de ejemplo que se han usado en el apartado anterior, durante la explicación del servidor local. De esta forma se pretende dar continuidad al ejemplo de lo que sería el flujo de trabajo estándar de un desarrollador que tiene el código alojado en GitLab y necesita interactuar con otros miembros del equipo.

4.2.1 Configuración a nivel administrador

Una vez instalada la aplicación como se ha expuesto al comienzo de esta sección, es posible realizar ciertos ajustes para personalizar esta herramienta. En este caso se va a seguir el flujo de trabajo que seguiría la persona del equipo de desarrolladores encargada de la administración de GitLab. En líneas generales esta configuración consiste en crear usuarios para el resto de los miembros del equipo, así como crear y configurar proyectos y grupos. También es posible realizar ciertas modificaciones visuales para adaptar la apariencia de GitLab a las necesidades del equipo.

La pantalla de configuración es accesible de dos formas diferentes, haciendo click en cualquiera de las áreas marcadas en la imagen se accede a la configuración:

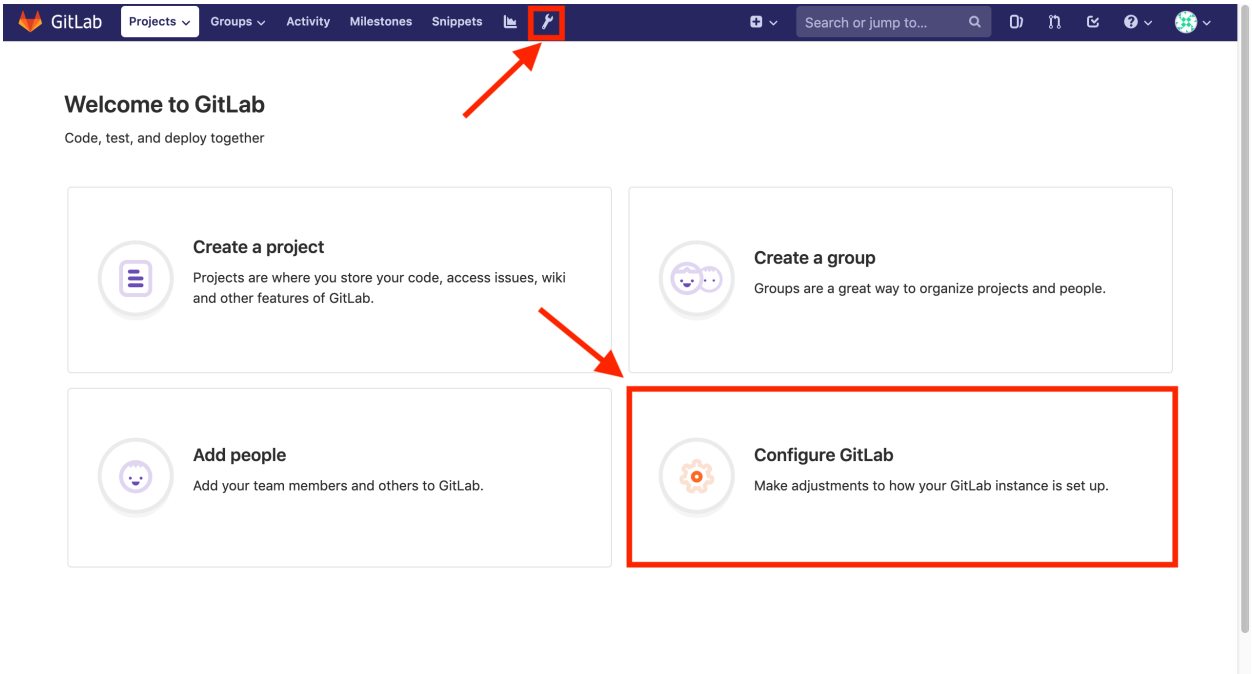

Ilustración 34 Botones/Enlaces de acceso a la configuración

Ambos enlaces llevan al mismo dashboard, una página donde aparece un resumen del estado actual de la aplicación: el número de proyectos, usuarios y grupos, así como información relativa a la propia aplicación y sus componentes:

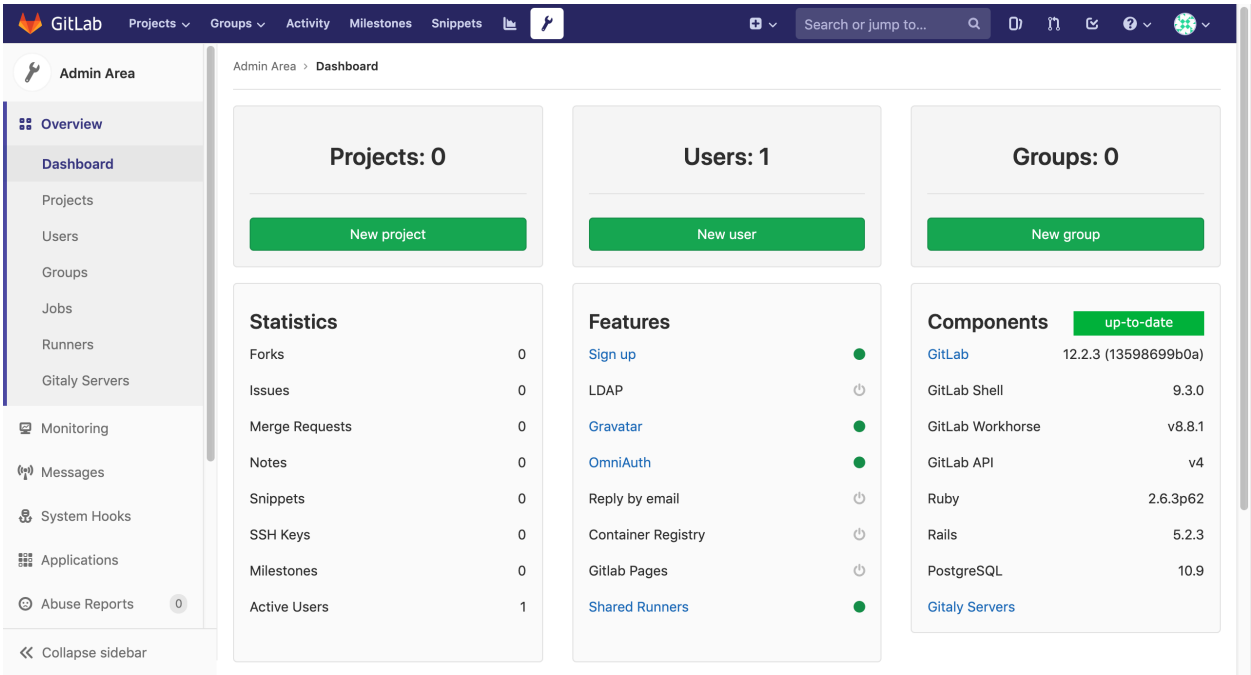

Ilustración 35 Página principal de configuración

En la parte izquierda aparece un menú desplegable con las diferentes opciones de configuración agrupadas por categorías. Las diferentes opciones se expanden una vez que se hace click en ellas, en la imagen anterior se

puede ver la sección "**Overview**" expandida, de la cual nacen una serie de subcategorías, en este caso la seleccionada es "Dashboard".

Otra pestaña útil dentro de la configuración que es solo de información es la sección de "**Monitoring**". En esta categoría se pueden encontrar diferente secciones, como "System Info" o "Logs" en las que se puede ver si hay alguna incidencia en la aplicación, para así proceder a subsanar el incidente.

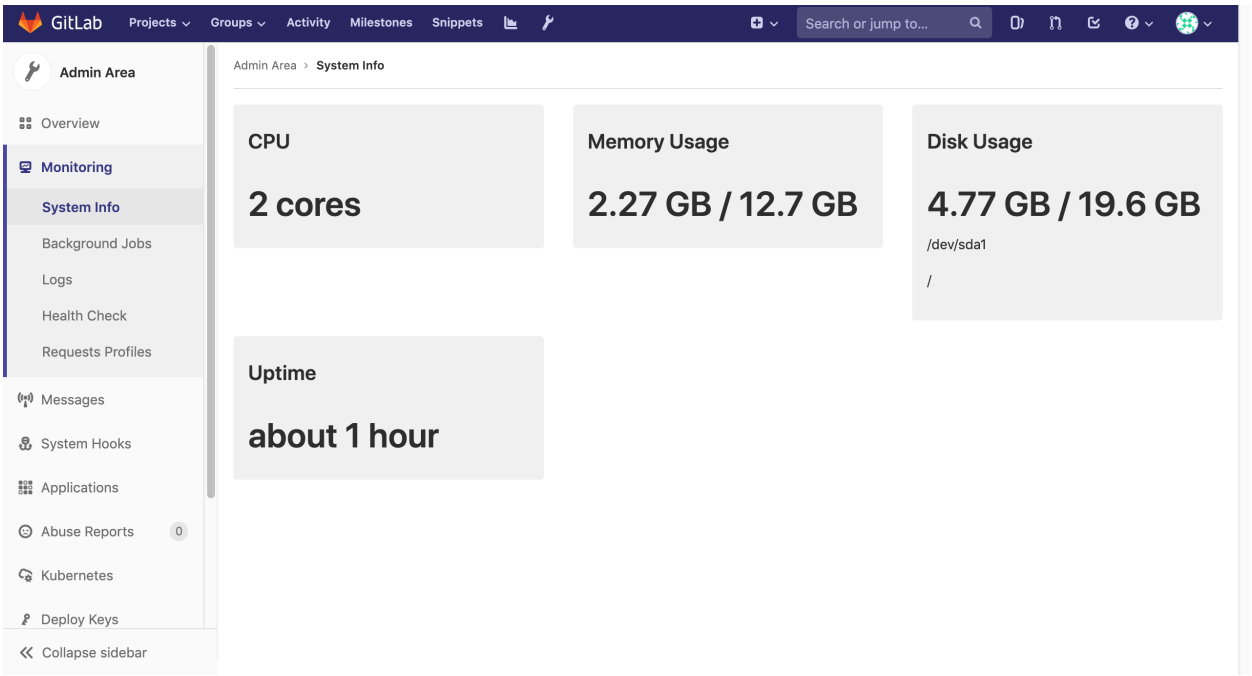

Ilustración 36 Información del sistema

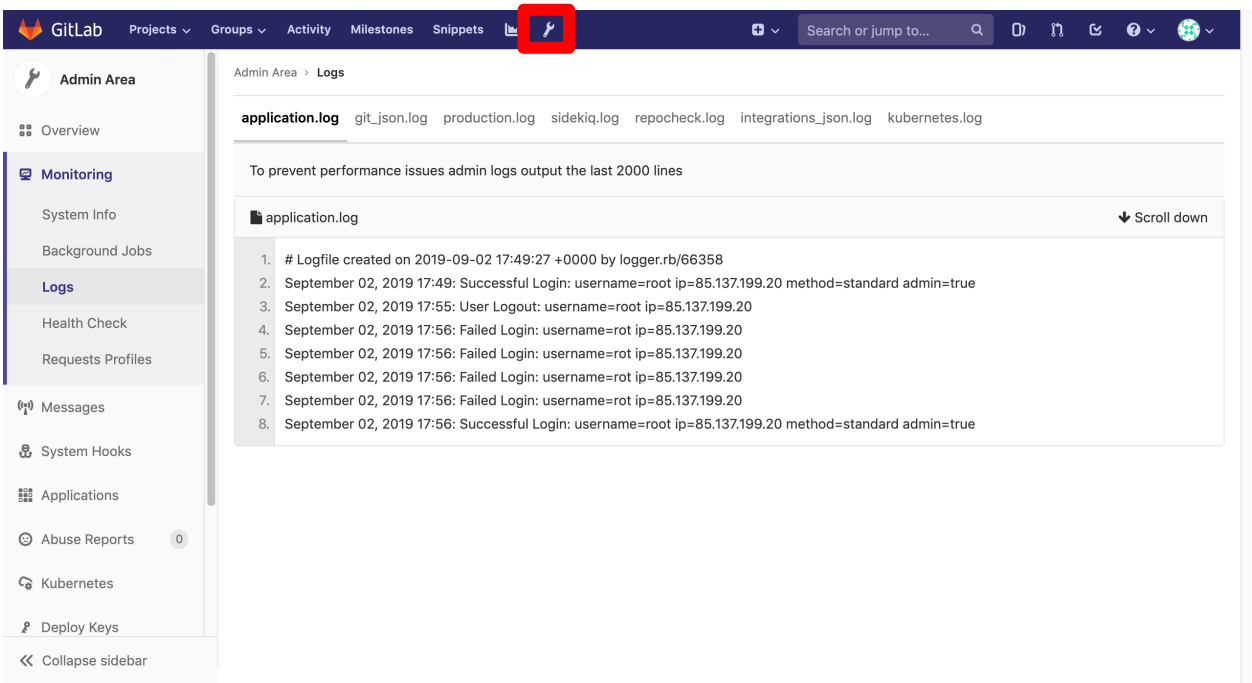

Ilustración 37 Logs de la aplicación e icono de acceso a las opciones

Desde la sección "**Admin Area**" accesible a través del icono de una llave inglesa es posible configurar un sinfín de opciones genéricas del funcionamiento de la aplicación, como el número máximo de proyectos

simultáneos en la aplicación, zona horaria, idioma, etc.

Usuarios

Desde la sección “Overview - Users” es posible gestionar los usuarios existentes en el sistema, así como crear nuevos usuarios haciendo click en “New user”.

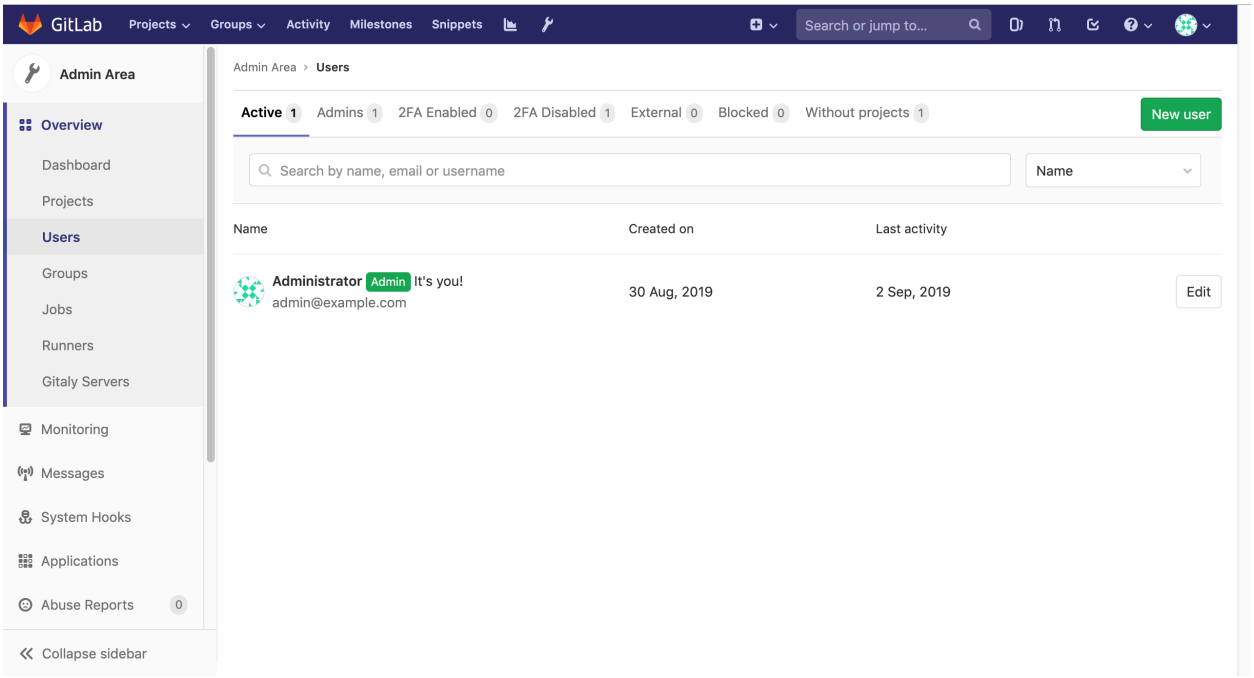

Ilustración 38 Página de configuración de usuarios

En este momento solo está creado el usuario “Administrador”, por tanto, se va a crear un nuevo usuario, para ello se hace click en el ya mencionado botón “New user”. A continuación, aparece una nueva página donde es necesario incluir la información de este nuevo usuario. Algunos campos como nombre, usuario o email son obligatorios, en cambio hay otras opciones de configuración que son opcionales como el avatar o la asociación de cuentas externas como Linkdin, Skype, etc.

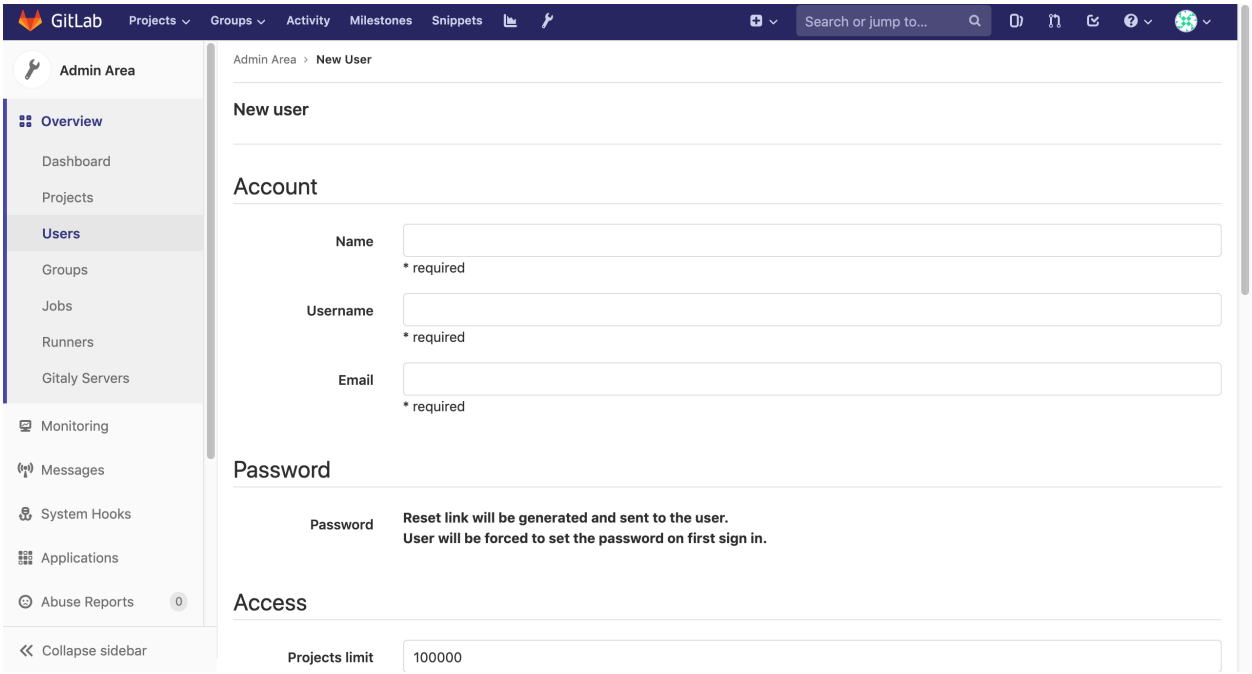

Ilustración 39 Nuevo usuario

Aparte de estos datos, hay otros de configuración que son importante mencionar. Por un lado, está el nivel de acceso, si es un usuario Regular, el cual tiene acceso a los proyectos y grupos a los que pertenezca o si es un usuario Admin, el cual tiene acceso a todos los grupos, proyectos y puede gestionar las diferentes características de la aplicación.

En este caso se va a crear un usuario regular, para ello se rellenan los diferentes campos y se hace click en "Create User". Una vez finalizado el proceso el sistema redirige hacia la página del usuario, desde donde es posible establecer una contraseña por defecto para el usuario que luego deberá modificar, en caso contrario el usuario recibirá un email con un enlace para su registro.

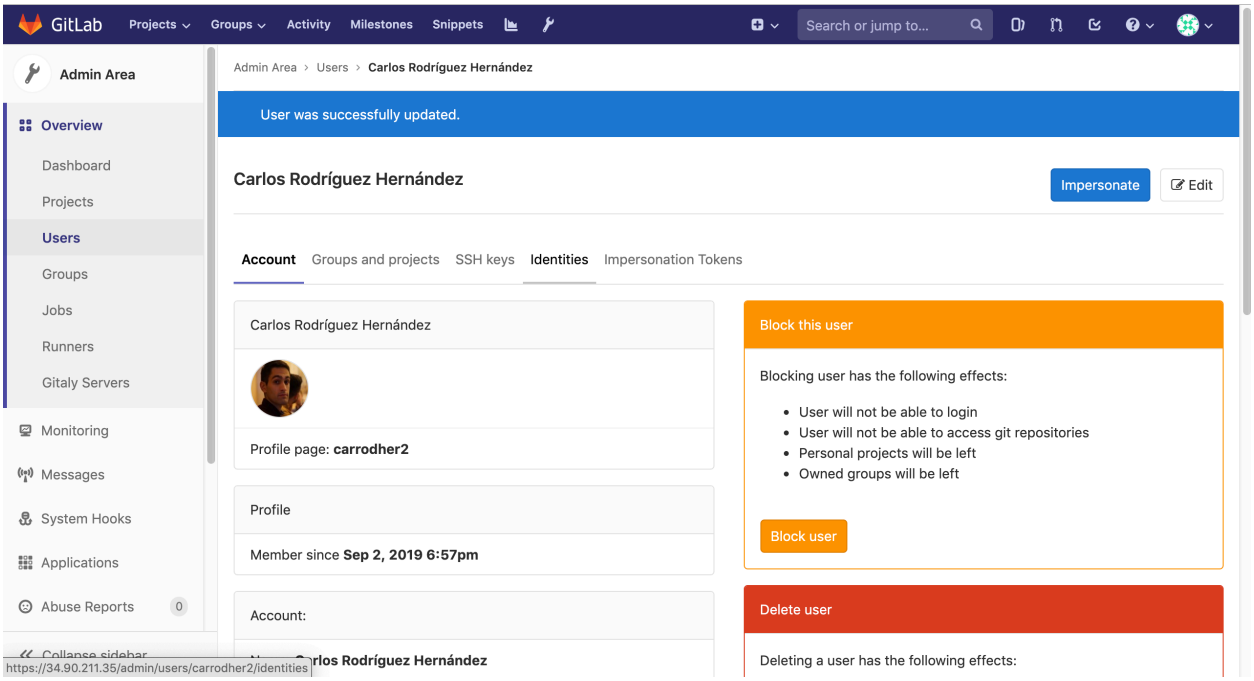

Ilustración 40 Usuario creado

Volviendo nuevamente a la página general de usuarios, ahora aparece este nuevo usuario aparte del Administrador que se vio anteriormente. Desde esta página es posible editar, eliminar o modificar los permisos de todos los usuarios.

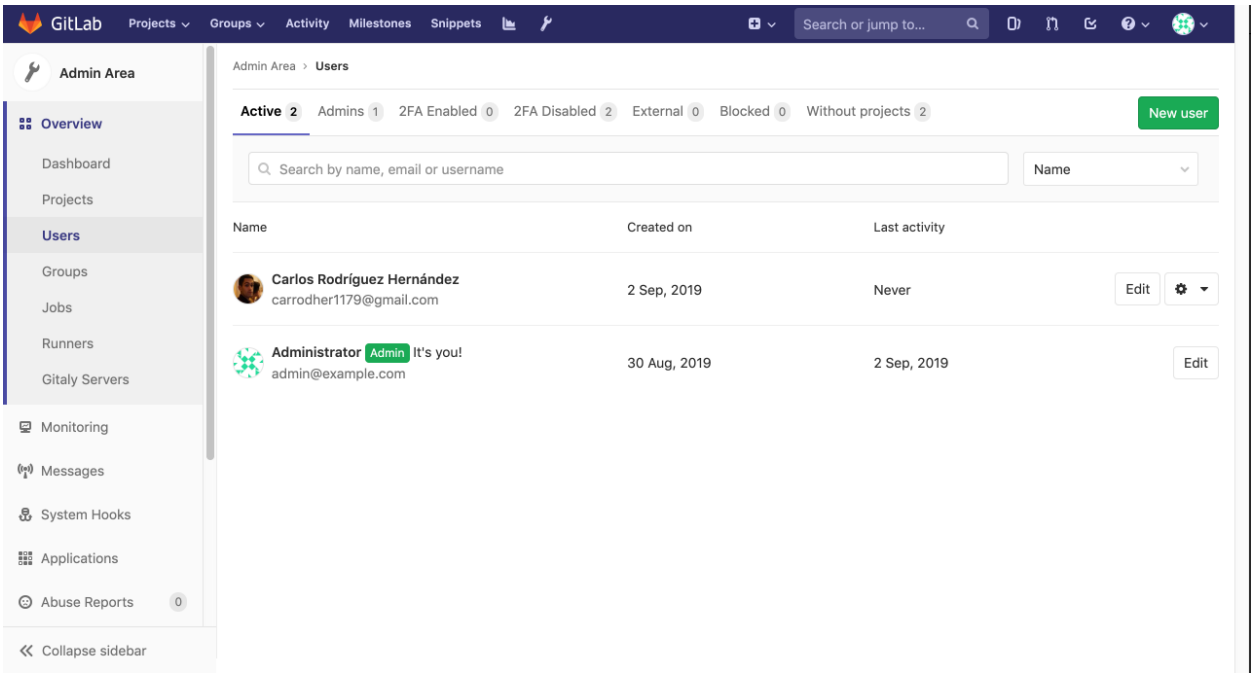

Ilustración 41 Página de configuración de usuarios con el usuario creado

Proyectos

Accediendo a la sección “Overview - Projects” es posible crear y organizar los proyectos. Un proyecto es la unidad fundamental en GitLab, donde se almacena el código de un repositorio, se gestionan las tareas asociadas a dicho proyecto y se realiza la revisión de código e incorporación de este al repositorio mediante “merge request” (solo en caso de que sea este el proceso elegido para incorporar código desde una rama a máster o producción).

Para crear un proyecto se hace click en “New project”, para dar paso a una nueva página donde se elige la forma de crear un proyecto, que puede ser: Crear un proyecto desde cero, es decir, vacío sin ningún tipo de contenido; crear el proyecto desde una plantilla, lo cual contiene la estructura básica de un proyecto en un lenguaje determinado; o importar un proyecto ya existente desde diferentes fuentes como puede ser GitHub.

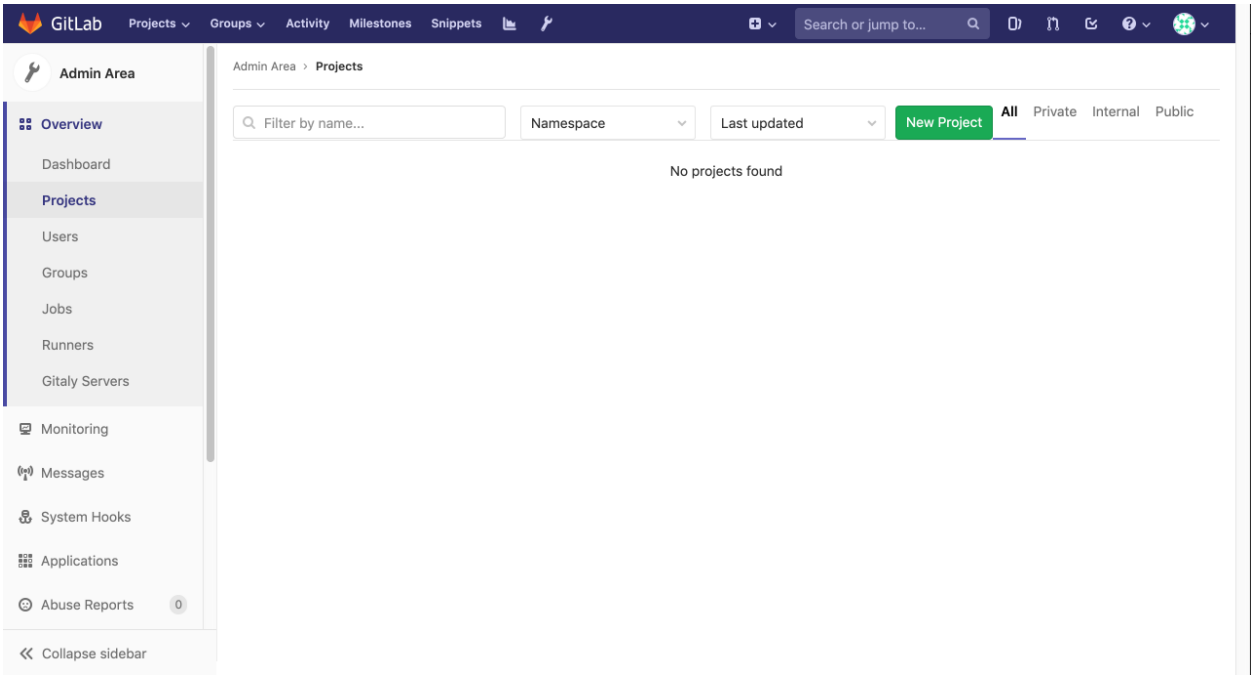

Ilustración 42 Página de configuración de proyectos

En este caso se va a crear un proyecto vacío llamado *backend* que posteriormente será añadido al grupo *desarrollo-web*. Grupos y proyectos pueden ser creados en cualquier orden, no es necesario seguir un orden establecido.

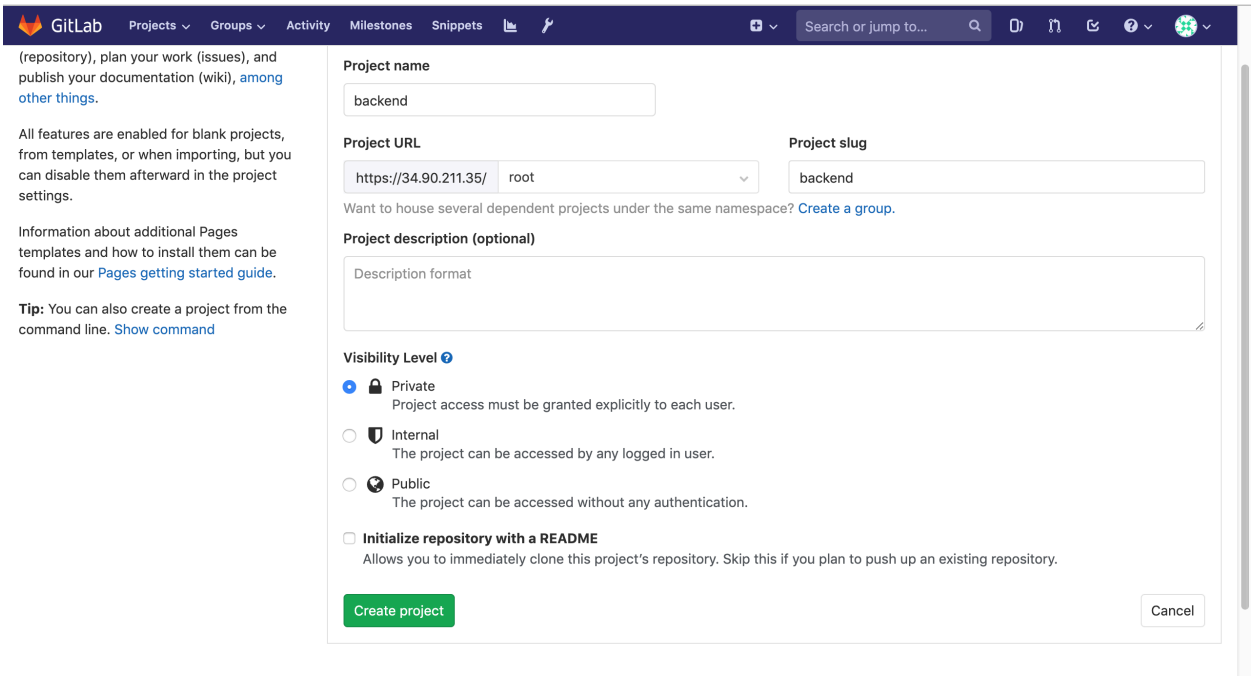

Ilustración 43 Nuevo proyecto

Como resultado de la creación del proyecto, aparece la página principal de dicho proyecto, donde se pueden observar las instrucciones para clonarlo mediante Git y añadir código, lo cual se desarrollará en el siguiente apartado, relativo al flujo de trabajo a nivel de usuario.

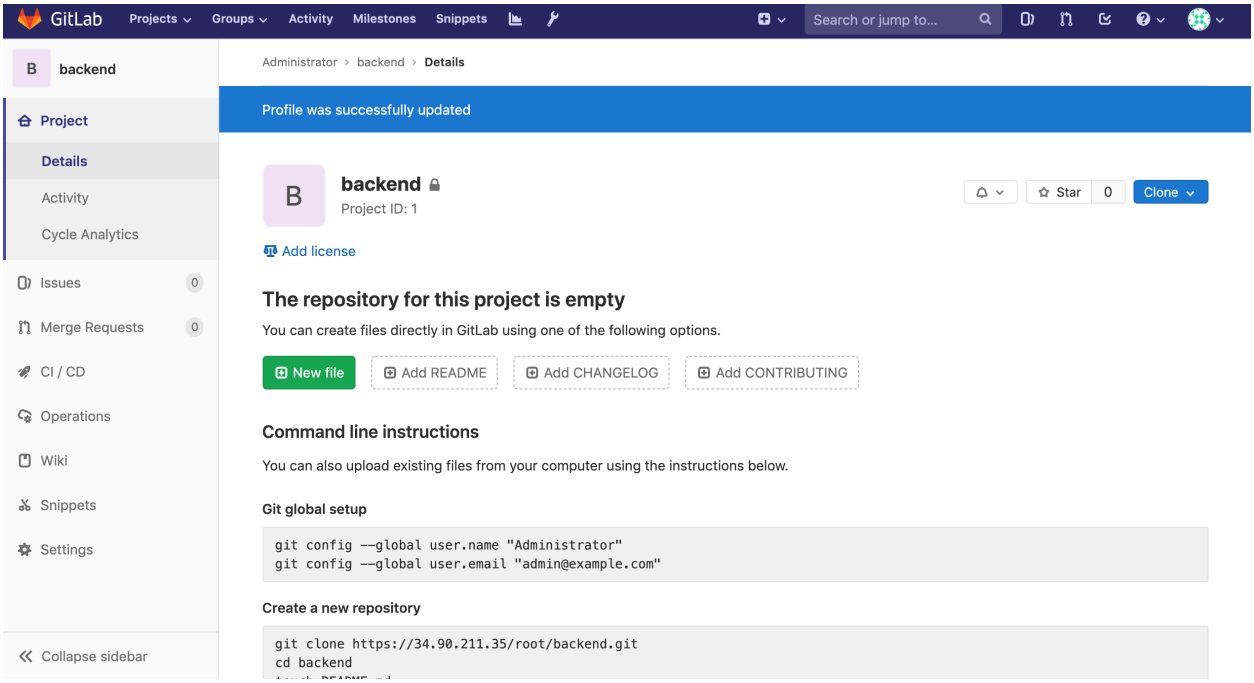

Ilustración 44 Proyecto creado

A nivel de administrador se pueden realizar diversos ajustes acerca del proyecto. Entre los ajustes que se pueden realizar a nivel de proyecto (también aplica a los grupos), uno de los más importantes es la visibilidad. Existen 3 niveles de visibilidad que establecen quién puede ver/interactuar con el proyecto:

- Proyectos públicos: Los proyectos públicos pueden ser clonados vía HTTPS por cualquier persona sin necesidad de autenticación.
- Proyectos internos: Los proyectos internos solo pueden ser clonados o vistos por los usuarios que tengan credenciales (usuario/contraseña) de acceso a GitLab.
- Proyectos privados: Los proyectos privados solo pueden ser clonados o vistos por los usuarios que son miembros del proyecto.

En general, para un grupo de trabajo lo normal es que los proyectos sean internos, es decir, cualquier usuario registrado en GitLab (con sus credenciales de acceso) puede interactuar con el proyecto. Los proyectos públicos están orientados para herramientas opensource, en los que la comunidad participa en el desarrollo, este concepto es similar a los repositorios públicos de GitHub.

Grupos

De manera similar, accediendo a la sección “Overview - Groups” es posible crear y organizar los grupos. Un grupo es una colección de diferentes proyectos con la ventaja de que se pueden realizar configuraciones para el grupo que aplican a diferentes proyectos. De la misma forma es posible asignar usuarios a un grupo y de esta forma otorgarle ciertos privilegios sobre ese grupo y los proyectos que contiene. Las reglas de visibilidad explicadas anteriormente para los proyectos también se pueden aplicar a nivel de grupo, lo cual seria aplicable a todos los proyectos englobados en el grupo.

Aparte de los permisos de los usuarios y la visibilidad, un grupo engloba todos los issues, merge requests y demás características de los proyectos como si se tratara de uno solo.

A modo de ejemplo, es habitual encontrar el caso de un equipo de trabajo que tiene diferentes roles formando subequipos como pueden ser: Administradores de sistemas, Desarrolladores Java y Desarrolladores gráficos. Cada uno de los sub-equipos de trabajo es independiente del resto y a priori trabajarán en proyectos diferentes. Se podría crear un grupo para cada subequipo donde se alojen los proyectos relativos al subequipo en cuestión.

También es posible orientarlo por producto en vez de por subequipos; en este caso si un grupo de

desarrolladores está trabajando en varios productos y cada producto tiene diferentes repositorios de código, lo cual es una buena práctica, se podría crear un grupo para cada producto que englobe los diferentes repositorios y proyectos. Se puede usar como ejemplo el desarrollo de una página web, lo cual sería el producto. Este producto tiene diferentes proyectos con sendos repositorios de código, por ejemplo, *frontend* y *backend*.

Obviamente esto es algo flexible, un mismo miembro puede pertenecer a diferentes grupos. También es posible crear subgrupos y formar diferentes niveles de anidación o jerarquía.

En la página inicial de grupos no aparece ninguno puesto que, por defecto, al igual que en los proyectos, no hay ninguno creado. Para crear uno hay que hacer click en "New group" y aparece una nueva página con los diferentes campos a rellenar para la creación de un nuevo grupo.

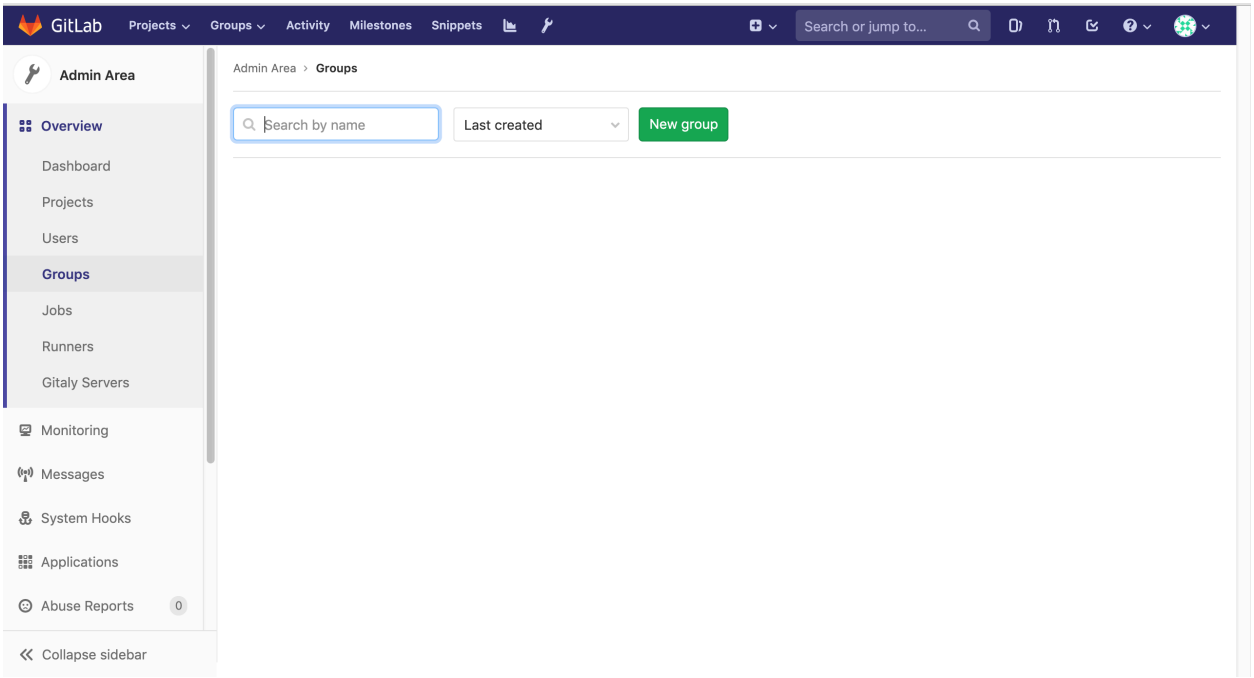

Ilustración 45 Página de configuración de grupos

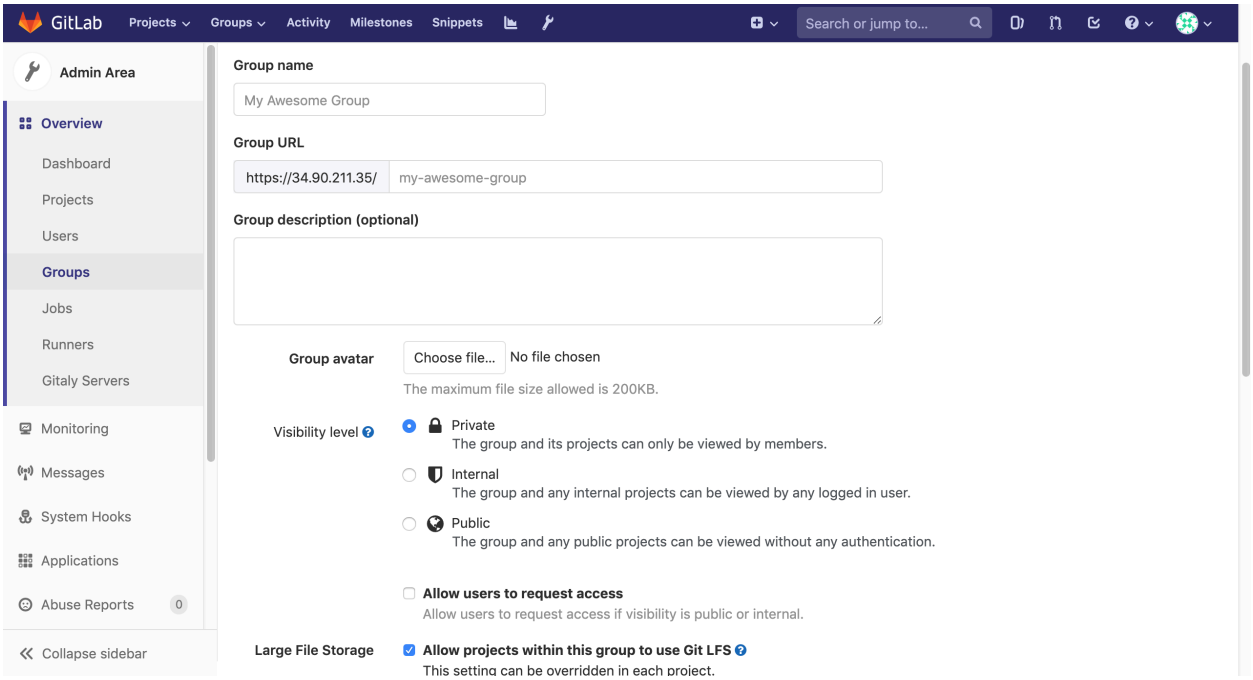

Ilustración 46 Nuevo grupo

Tras la creación del grupo aparece la página del grupo en cuestión, desde la cual es posible añadir miembros al grupo y realizar algunos ajustes básicos.

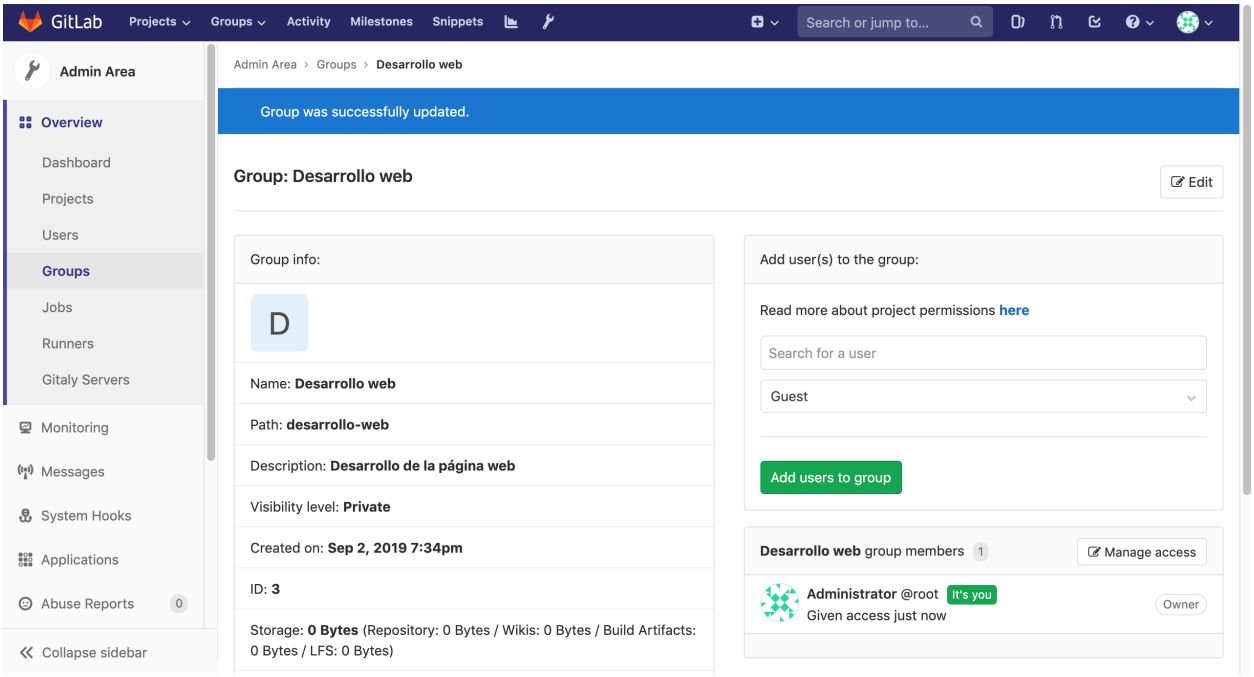

Ilustración 47 Grupo creado

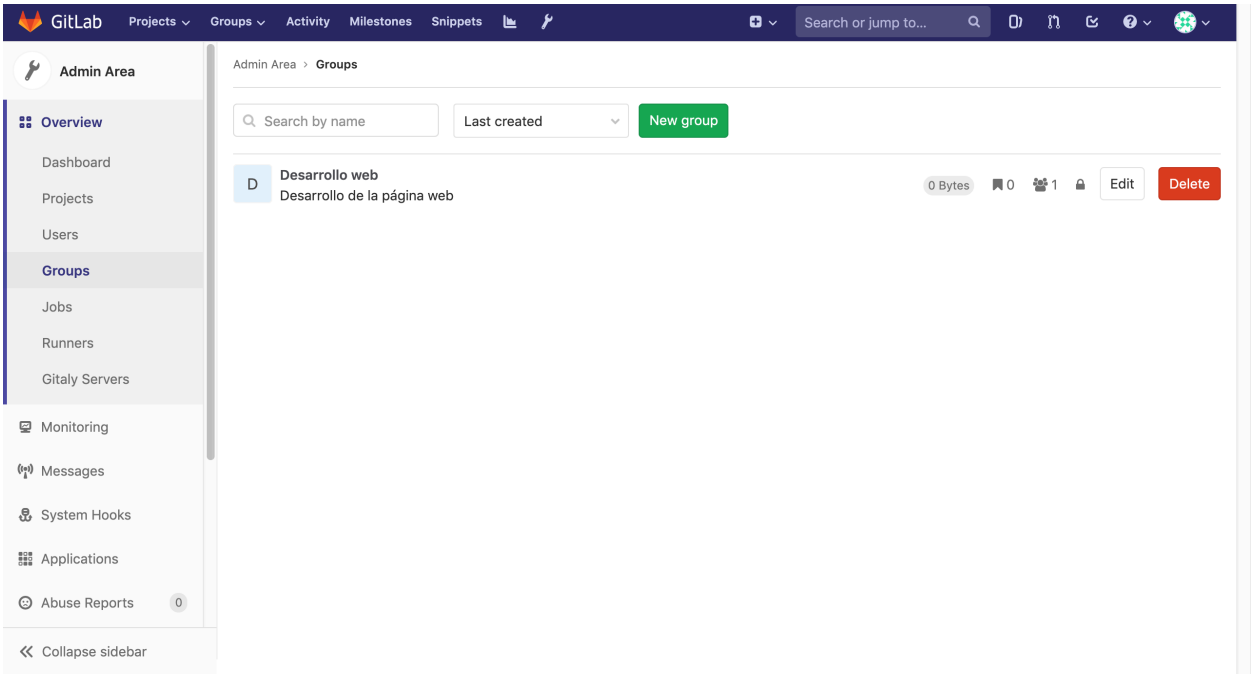

Ilustración 48 Página de configuración de grupos con el grupo creado

Personalizaciones visuales

En la sección "**Appearance**" es posible configurar diferentes aspectos visuales de la aplicación, de esta forma se puede personalizar la apariencia de la aplicación para hacerla más corporativa y que el equipo trabaje usando una aplicación que esté alineada visualmente con la marca para la que trabaja. Desde esta sección es posible cambiar la barra de navegación, el icono que aparece en el navegador, la cabecera o pie de página entre otras cosas.

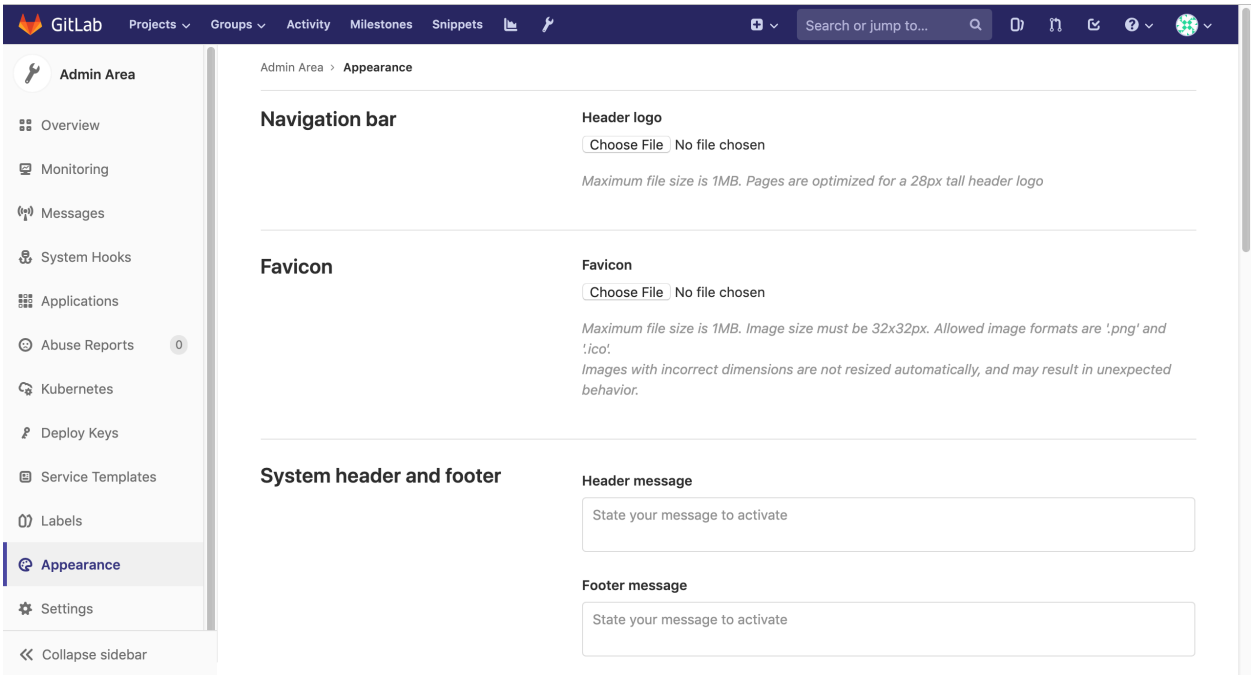

Ilustración 49 Configurar apariencia

En las secciones anteriores se han visto una serie de configuraciones iniciales que pueden ser realizadas por el

usuario administrador de la aplicación, así como la manera de crear usuarios, proyectos y grupos, que son las unidades más importantes con las que trabaja GitLab.

Como se ha mencionado con anterioridad, GitLab ofrece un sinfín de opciones configurables, para profundizar en ellas más allá de lo mencionado a lo largo de esta sección es recomendable acudir a la documentación oficial disponible en la página web de la herramienta [46].

4.2.2 Flujo de trabajo a nivel de usuario

Todo lo visto en el apartado anterior son acciones de configuración que deben ser llevadas a cabo por los administradores de GitLab. Algunas de ellas pueden ser realizadas por usuarios regulares, como es el caso de la creación de proyectos.

En este apartado se va a ver el flujo de trabajo habitual de un desarrollador usando el entorno de desarrollo basado en Docker y GitLab. Se da por hecho que el administrador ha creado una cuenta para un segundo usuario (carrodher2) como se ha visto en el apartado anterior. Este usuario accede a la interfaz web de GitLab usando la contraseña que le ha asignado el administrador e inmediatamente, por motivos de seguridad, aparece una pantalla para que la cambie y establezca su propia contraseña.

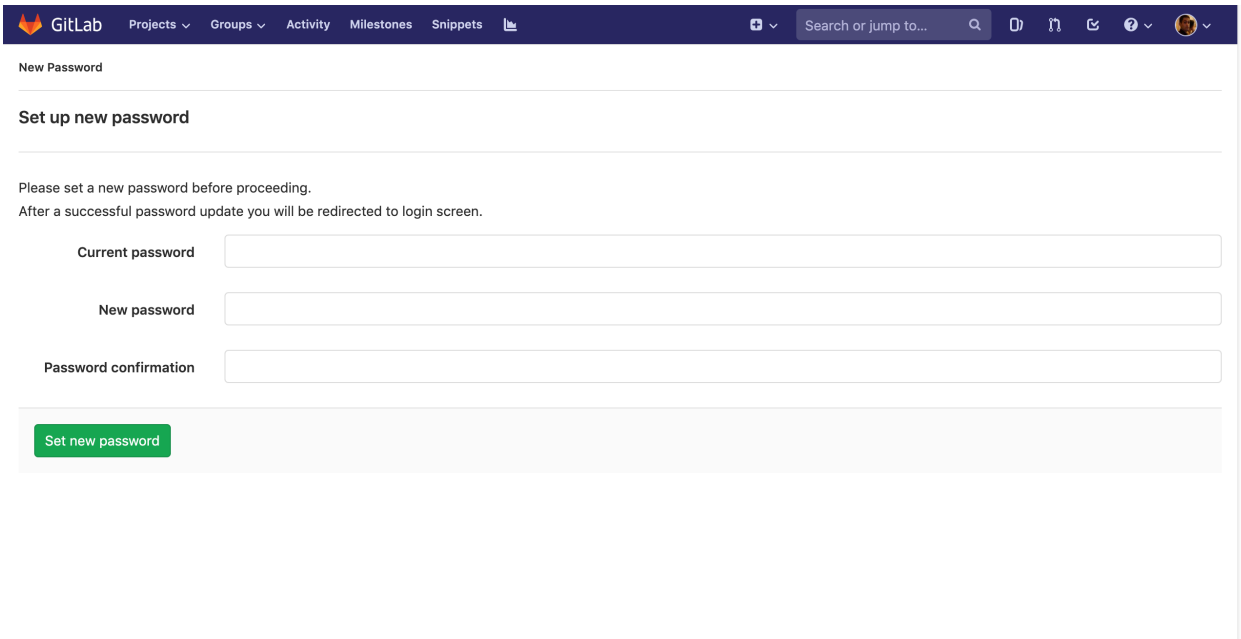

Ilustración 50 Primer acceso de un usuario regular

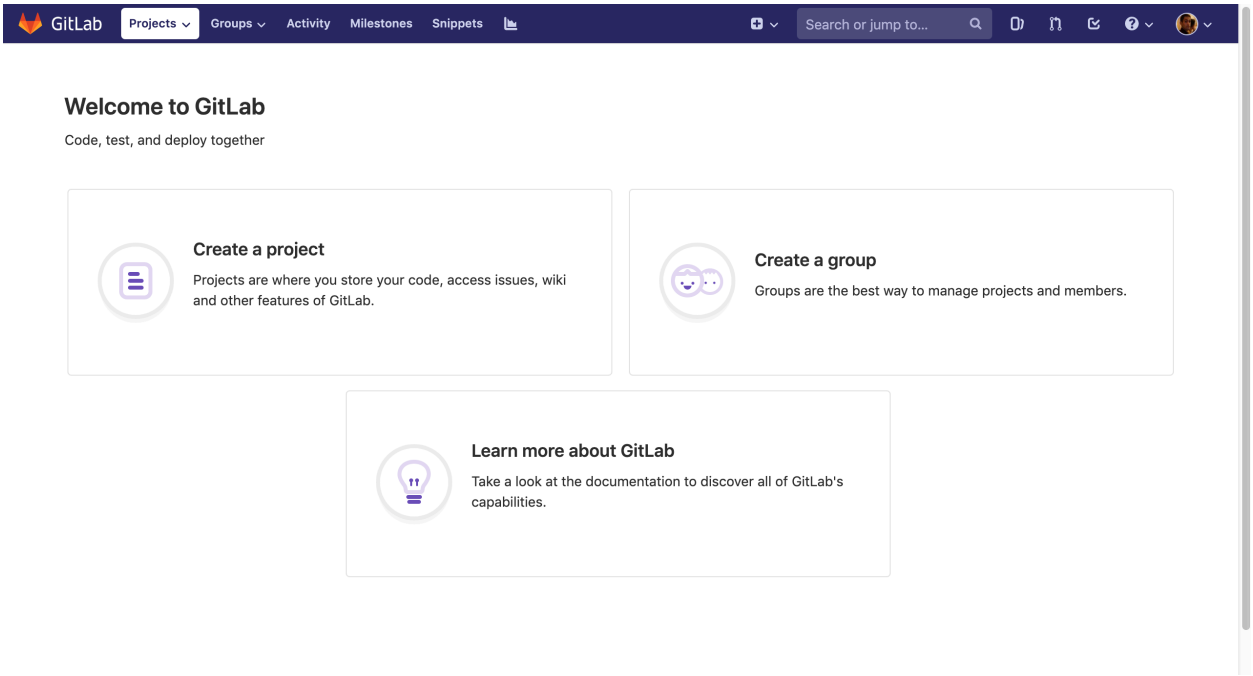

Ilustración 51 Página principal tras el acceso

Como se puede ver en la imagen anterior, el usuario regular no cuenta con los enlaces para acceder a la configuración que aparecían en el usuario administrador. Sí es posible cambiar algunos ajustes a nivel de usuario accediendo a "Settings" desde el menú desplegable que aparece al hacer click en el avatar del usuario.

Proyectos

Como se puede apreciar en la imagen anterior, este usuario puede crear proyectos y grupos. En esta sección se van a crear dos proyectos usando dos métodos diferentes.

Crear proyecto y añadir código existente

El primero de ellos será creando un proyecto vacío y posteriormente añadiendo el código a él.

Conviene aclarar que al referirse a crear un "proyecto vacío" es a nivel de GitLab, de manera inicial. El desarrollador tendrá su código alojado en su máquina local o en otro repositorio y quiere que dicho código empiece a formar parte de Gitlab. En el caso de que se desee crear un proyecto nuevo, sin ningún código existente previamente, el proceso es el mismo, pero en vez de copiar los ficheros existentes se empieza a desarrollar directamente en este nuevo proyecto desde el comienzo.

Este será el proceso habitual si ya tenemos un repositorio con código existente (en local o en otro repositorio, GitHub, por ejemplo). Para ello se crea el proyecto vacío, se clona el nuevo repositorio vacío desde GitLab a la carpeta *projects* de nuestro entorno local y se copian los ficheros existente en nuestra máquina local a esta nueva localización.

```
$ docker run -e USER_NAME='Carlos Rodriguez' -e USER_EMAIL=user@example.com -v ~/entorno-de-
desarrollo/projects:/root/projects -it carrodher/entorno-de-desarrollo:0.0.1

root:~
▶ ls -la projects
total 4
drwxr-xr-x 4 root root  128 Sep  1 17:05 .
drwx------ 1 root root 4096 Sep  3 05:51 ..
drwxr-xr-x 5 root root  160 Sep  1 17:17 sample-PHP-app
drwxr-xr-x 9 root root  288 Sep  1 17:05 sonarqube-JS
```

Bloque de código 35 Acceso al entorno de desarrollo y comprobación de los proyectos existentes

Como se puede ver en el bloque de código anterior, accediendo al entorno de desarrollo se pueden ver los dos proyectos en los que está trabajando el desarrollador.

Para incluir este proyecto/repositorio en GitLab lo primero que hay que hacer es crear un proyecto vacío en GitLab, para ello se sigue un proceso similar al explicado anteriormente para el usuario administrador. Se hace click en "Create project" y se rellenan los campos requeridos, así como alguna opción de configuración que se quiera modificar. El nombre elegido será el mismo.

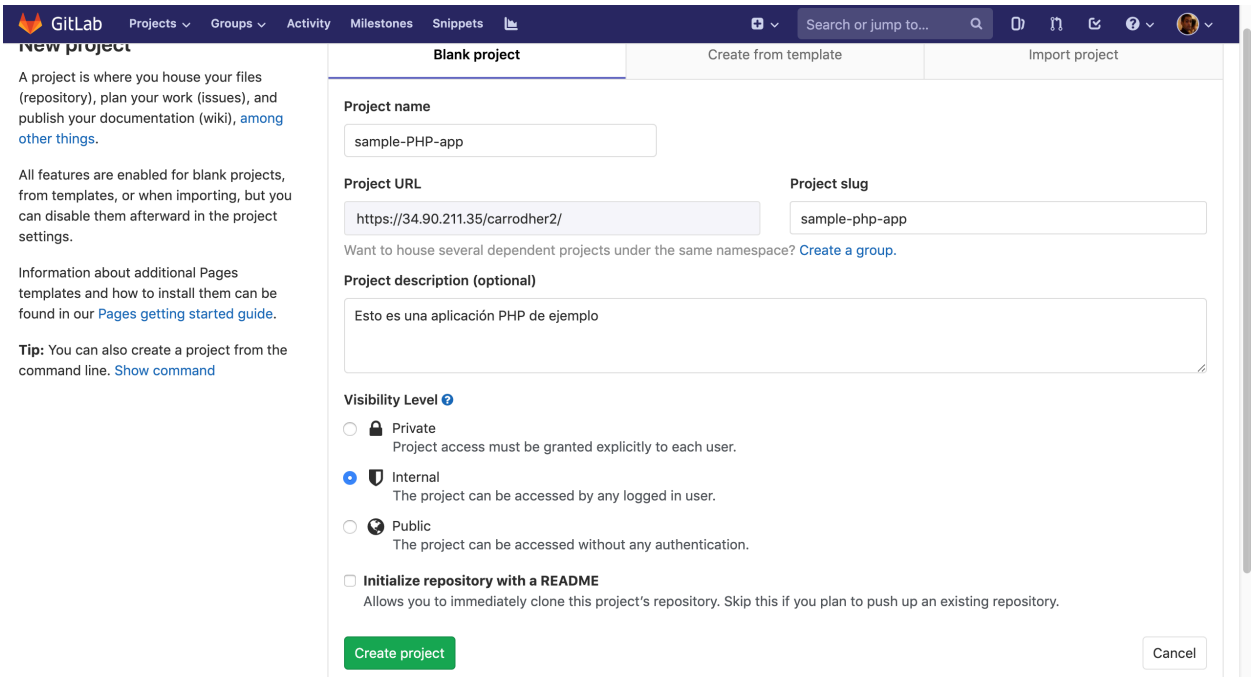

Ilustración 52 Crear nuevo proyecto

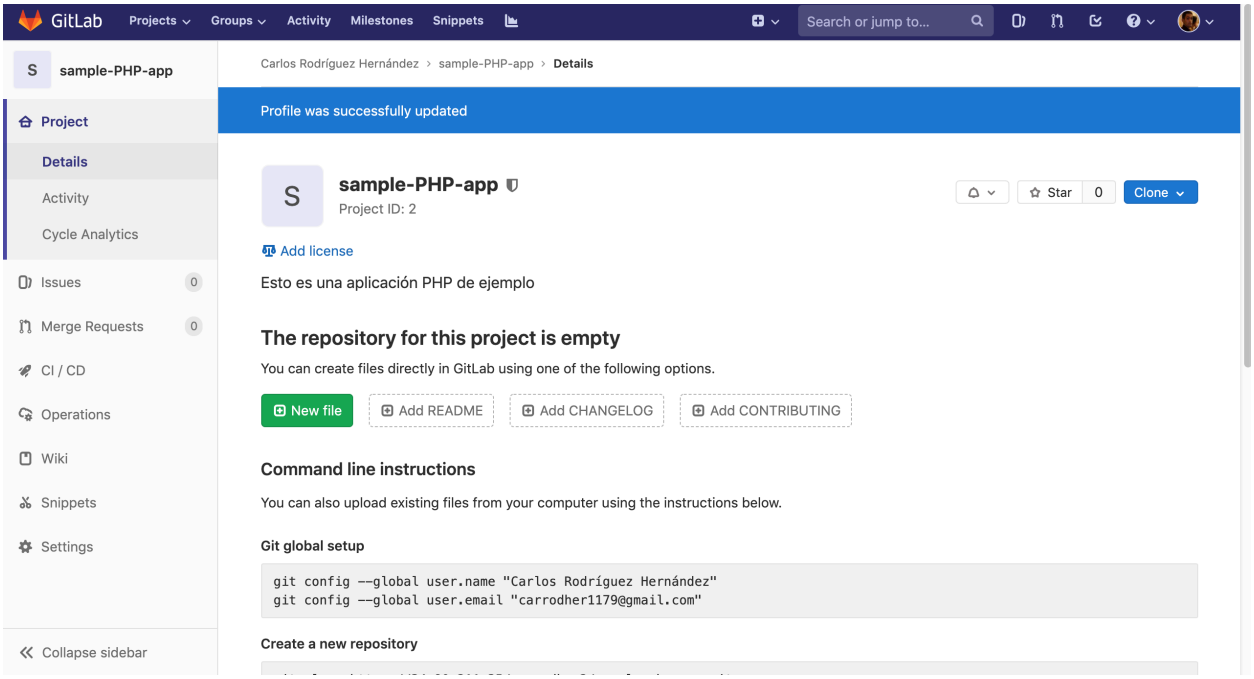

Ilustración 53 Proyecto *sample-PHP-app* creado

Como se puede ver en la imagen anterior, el proyecto se ha creado sin contenido y la propia interfaz web de GitLab ofrece la posibilidad de crear nuevos ficheros, añadir README, CHANGELOG etc. Se podría realizar algunas de estas acciones para añadir contenido al proyecto, pero cabe recordar que el proyecto ya existe en nuestro entorno local, por lo tanto, lo que se pretende es copiar los ficheros que ya existen a este nuevo repositorio para que aparezcan en GitLab.

En la imagen previa aparecen algunas instrucciones para configurar Git en nuestra máquina local, como el entorno de desarrollo está preconfigurado, no sería necesario realizar ninguna acción más allá de asegurarse que las variables de entorno `USER_NAME` y `USER_EMAIL` que se pasaban en el comando `docker run` coincidan con las credenciales usadas para acceder a GitLab.

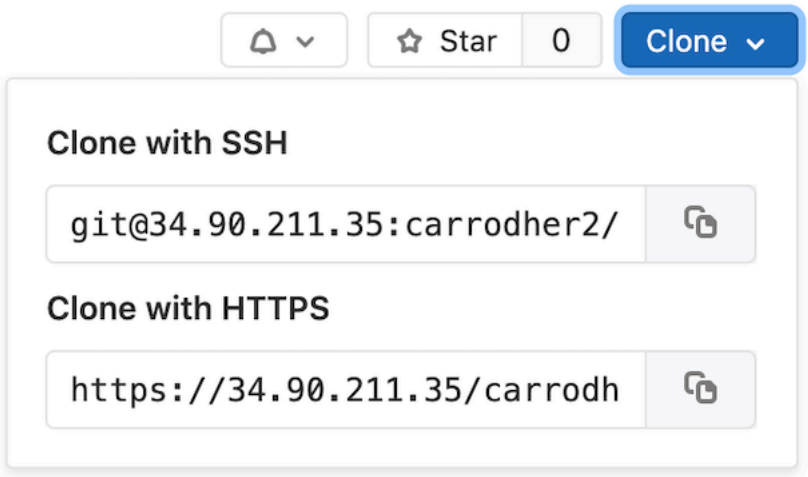

Ilustración 54 Detalle del botón "Clone"

De igual manera, en cada proyecto aparece un botón “Clone” con la dirección del proyecto en cuestión. Esta dirección se usa para hacer pull y push desde el entorno de desarrollo. Esta dirección se usará en varios

ejemplos descritos en los siguientes párrafos. Nótese que la dirección IP que aparece, como es de esperar, es la IP de la máquina en la que se ejecuta GitLab.

En este punto pueden darse dos circunstancias, la primera de ellas es que el código que queremos añadir a GitLab y que ya existe en nuestra máquina local estuviera siendo manejado por Git, es decir, aunque no ha sido subido a GitHub u otro repositorio de código, se estaba usando Git para control de versiones. La otra alternativa es que el código haya sido desarrollado sin seguir control de versiones, simplemente se han desarrollado los diferentes ficheros relativos al código propiamente dicho.

En cualquiera de los casos es simple añadir este proyecto existente a GitLab y a partir de este momento gestionarlo a través de la herramienta mediante Git.

- Proyecto existente con Git

  Si el proyecto existe y los ficheros han sido añadidos al control de versiones es útil mantener toda la historia de Git, por ello se realiza el siguiente proceso:

```
$ docker run -e USER_NAME='Carlos Rodriguez Hernandez' -e USER_EMAIL=carrodher1179@gmail.com -v ~/entorno-de-
desarrollo/projects:/root/projects -it carrodher/entorno-de-desarrollo:0.0.1

root:~
▶ cd projects/sample-PHP-app

root:~/projects/sample-PHP-app
▶ git remote rename origin old-origin && \      # Renombra el registro remoto existente
git remote add origin https://34.90.211.35/carrodher2/sample-php-app.git && \ # Añade un nuevo registro remoto
git config credential.helper store && \         # Permite que las credenciales de GitLab solo se pidan una vez
git push -u origin --all && \                   # Hace push de todas las ramas locales
git push -u origin --tags                       # Hace push de todas las etiquetas locales
```

Bloque de código 36 Acceso al entorno y mover a GitLab proyecto existente

En los comandos del bloque de código anterior, lo primero que se hace es acceder al entorno de desarrollo como se ha visto en el apartado anterior, teniendo en cuenta que el nombre y email usados son los mismos que se ha configurado en GitLab. Posteriormente se accede al proyecto con Git que se quiere mover al nuevo proyecto creado en GitLab y se ejecutan una serie de comandos para que a partir de este momento todos las acciones de Git sean enviadas a GitLab.

Git por defecto antes de enviar los cambios mediante `git push` a un proyecto privado o interno solicita las credenciales de GitLab (usuario/contraseña que se usa para acceder), mediante el comando `git config credential.helper store` solo será necesario introducirlo en una ocasión. También es posible configurar una clave SSH y clonar el repositorio usando este protocolo en vez de HTTPS.

Tras estos comandos y recargando la interfaz web, aparecen los ficheros, así como toda la información relativa a Git:

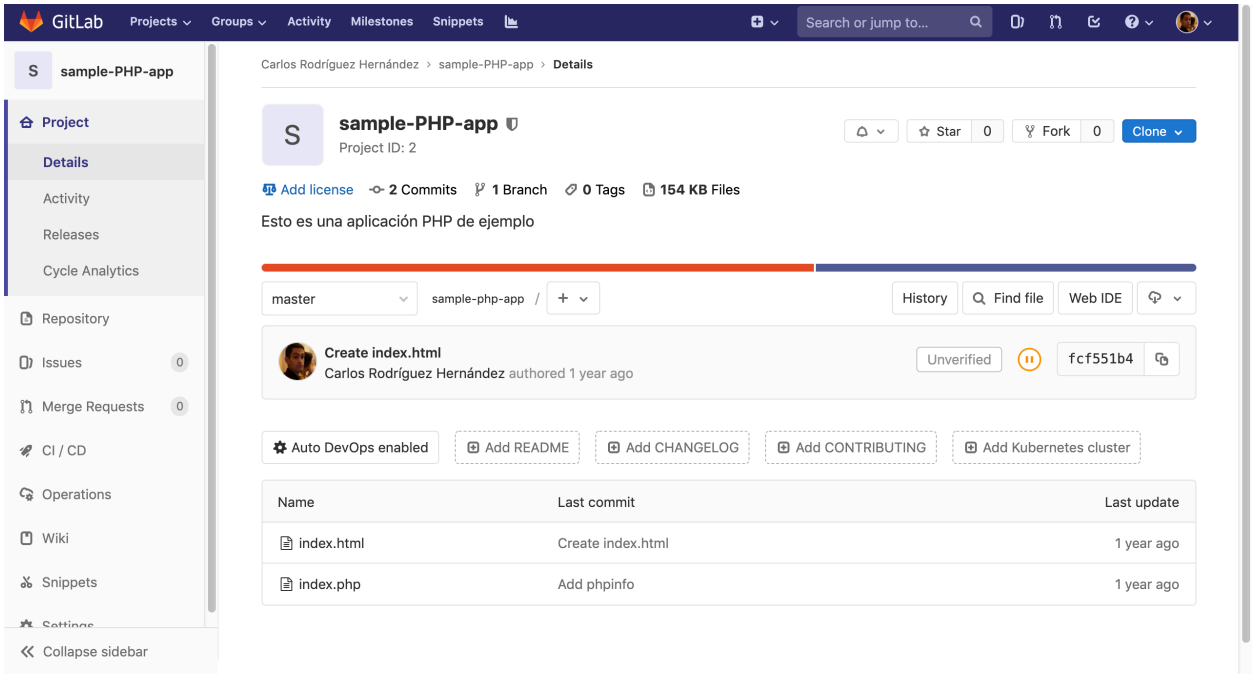

Ilustración 55 Proyecto existente con Git añadido a GitLab

- Proyecto existente sin Git

  En el caso de que el proyecto que se quiera añadir no estuviera siendo manejado por un sistema de control de versiones como Git no hay ningún histórico que mantener, simplemente se inicializará Git y se subirá el código a GitLab como un primer y único commit; desde este momento será posible seguir un flujo de trabajo basado en Git.

```
$ docker run -e USER_NAME='Carlos Rodriguez Hernandez' -e USER_EMAIL=carrodher1179@gmail.com -v ~/entorno-de-
desarrollo/projects:/root/projects -it carrodher/entorno-de-desarrollo:0.0.1

root:~
▶ cd projects/sample-PHP-app

root:~/projects/sample-PHP-app
▶ git init && \                                  # Inicializa este proyecto como un repositorio Git
git remote add origin https://34.90.211.35/carrodher2/sample-php-app-2.git && \ # Añade un nuevo registro remoto
git add . && \                                   # Añade los cambios actuales (todo)
git commit -m "Initial commit" && \              # Realiza el primer commit
git push -u origin master                        # Sube los cambios al repositorio de GitLab (apuntado por origin)
```

Bloque de código 37 Mover a GitLab un proyecto existente sin Git

En este caso, el repositorio aparece con un solo commit con la fecha actual (Just now), no como en el caso anterior que se pueden observar diferentes commits con la fecha de creación del repositorio original, puesto que mantiene la historia de Git.

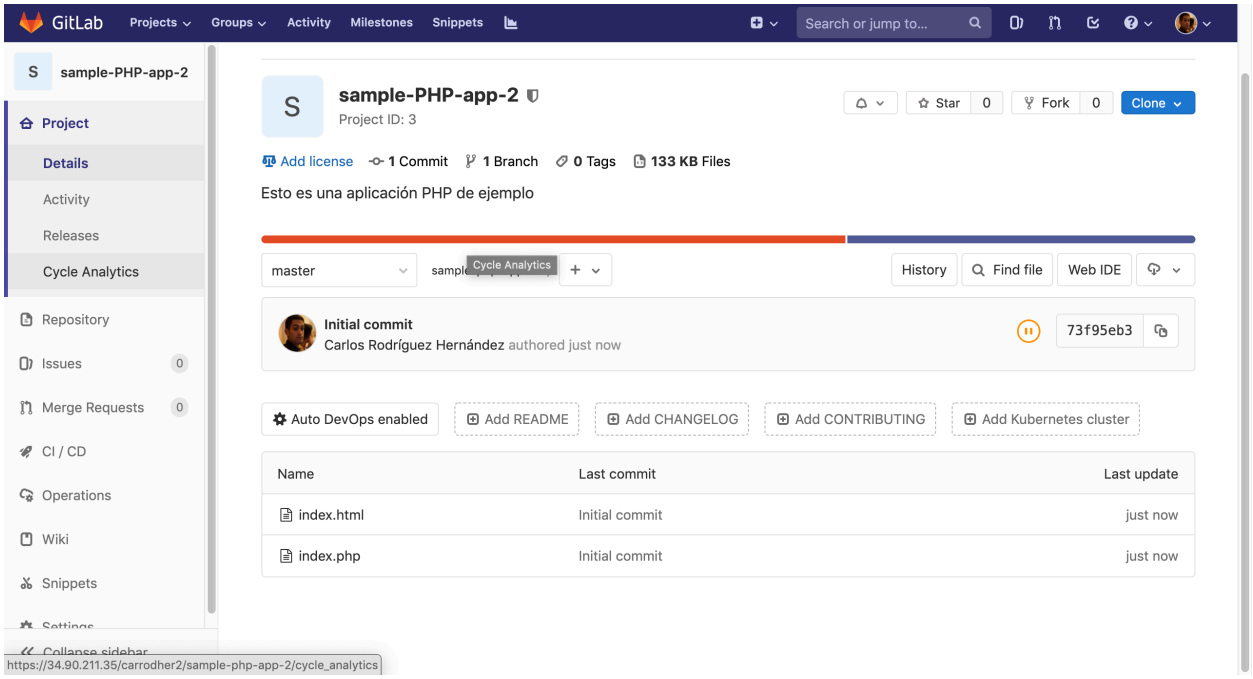

Ilustración 56 Proyecto existente sin Git añadido a GitLab

De esta forma es posible crear proyectos en GitLab basados en proyectos existente de antemano, ya sea con Git o sin él.

Importar proyecto desde GitHub

Otra opción es añadir un repositorio existente en GitHub, el resultado sería el mismo que se ha visto en el punto anterior para la versión con Git, puesto que se mantiene todo el histórico del control de versiones; pero es algo más fácil de llevar a cabo puesto que se pueden importar varios proyectos de una misma cuenta u organización de GitHub a la vez. Esto es muy útil para cuando se quieren importar todos los proyectos de un equipo de trabajo o no se tiene el repositorio clonado en el equipo local.

Para que GitLab tenga acceso a la información de GitHub hay que crear un token en GitHub con permisos de lectura sobre los repositorios. Una vez creado el token en GitHub siguiendo las instrucciones, se pega en GitLab y ya es posible listar los diferentes repositorios que se pueden importar.

Para crear un token en GitHub hay que acceder a la configuración del usuario a través de la interfaz web, para ello basta con hacer click en “Settings” que aparece en el menú desplegable:

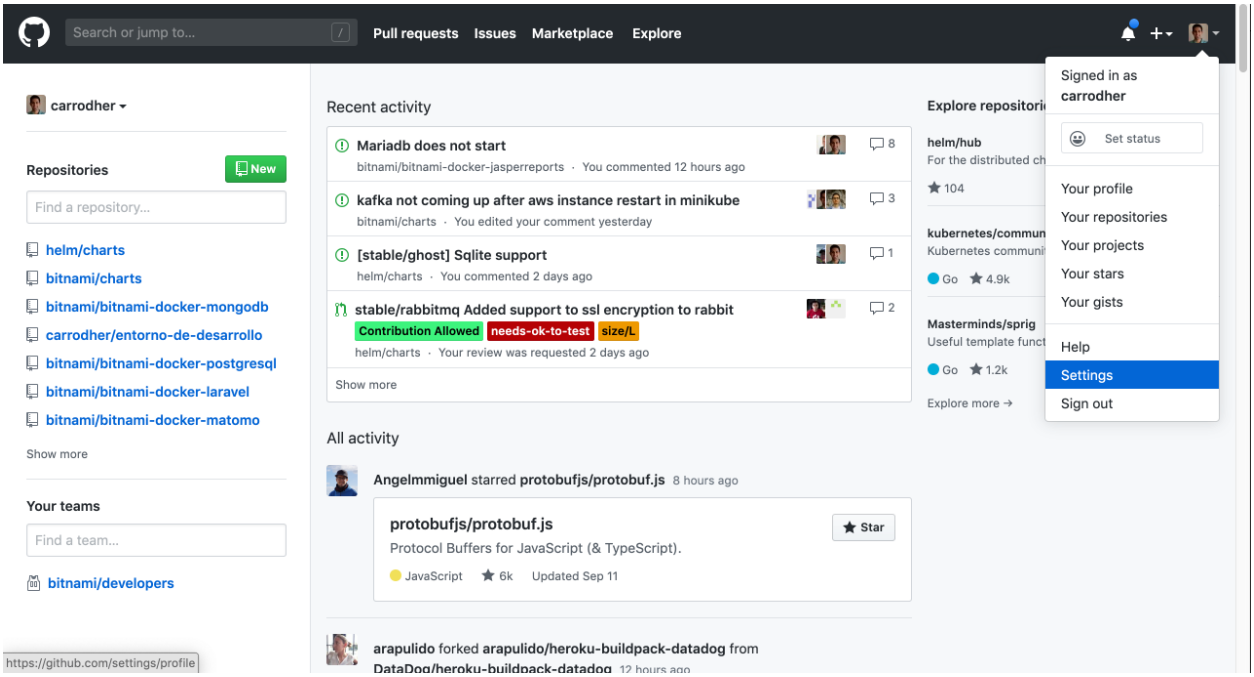

Ilustración 57 Configuración de GitHub

En la nueva página que aparece, es necesario hacer click en "Developer settings", tras lo cual aparece la siguiente pantalla, donde aparece la opción "Personal access tokens". En esta pantalla es posible generar un nuevo token con los permisos que se deseen configurar:

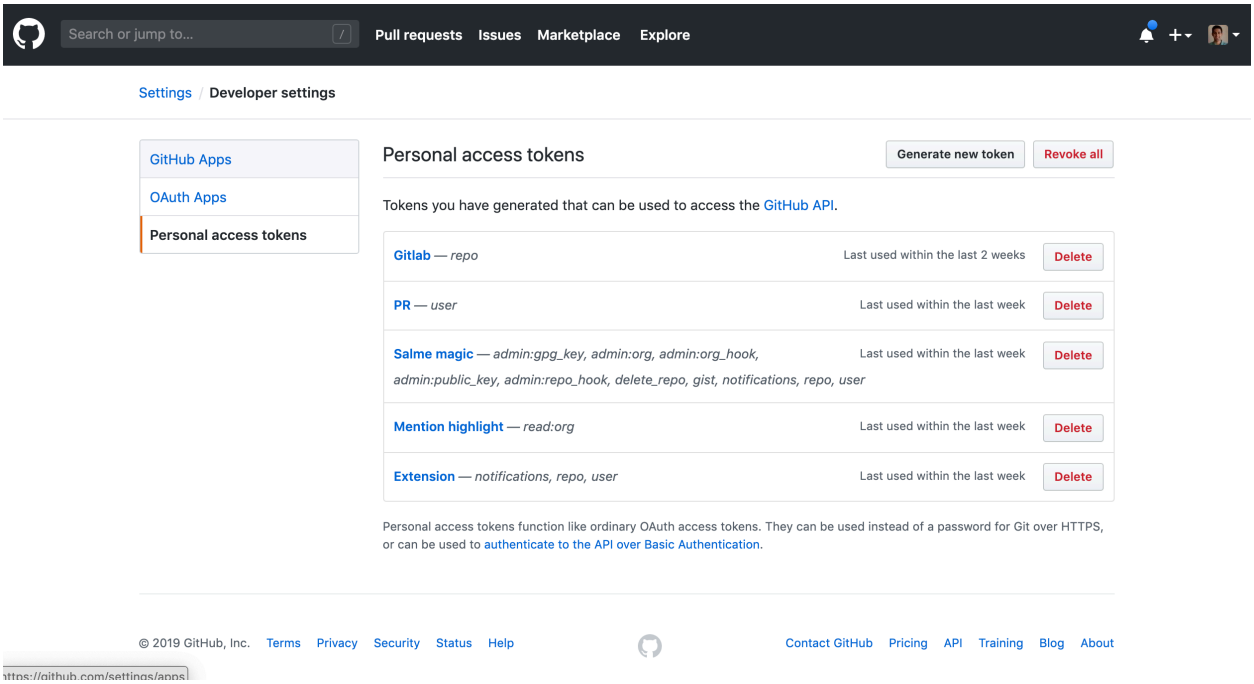

Ilustración 58 Crear token en GitHub

Haciendo click en el botón de generar el nuevo token aparece un formulario en el que debemos elegir un nombre para este token, así como el ámbito al que queremos dar acceso para dicho token, en este caso, tal y como especifica la página de GitLab para importar el proyecto, es suficiente con dar permisos para acceder a los repositorios:

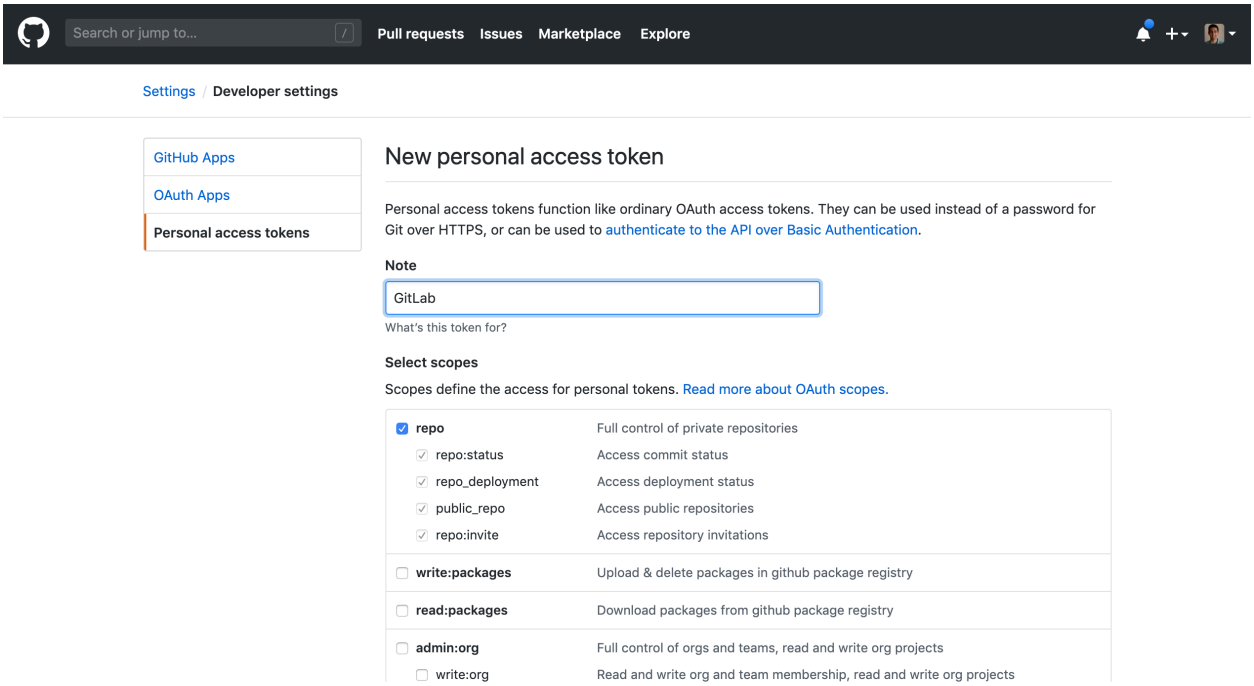

Ilustración 59 Crear token con permisos sobre repositorios

Una vez generado el token es importante copiarlo, puesto que es el parámetro que hay que pegar en GitLab. Esta es la forma de permitir que GitLab acceda a GitHub, mediante un token único, generado por el usuario registrado en ambas plataformas mediante su contraseña y solo con acceso a aquello que es necesario, en este caso, puesto que lo que se desea es importar repositorios, solo se ha garantizado acceso a los repositorios.

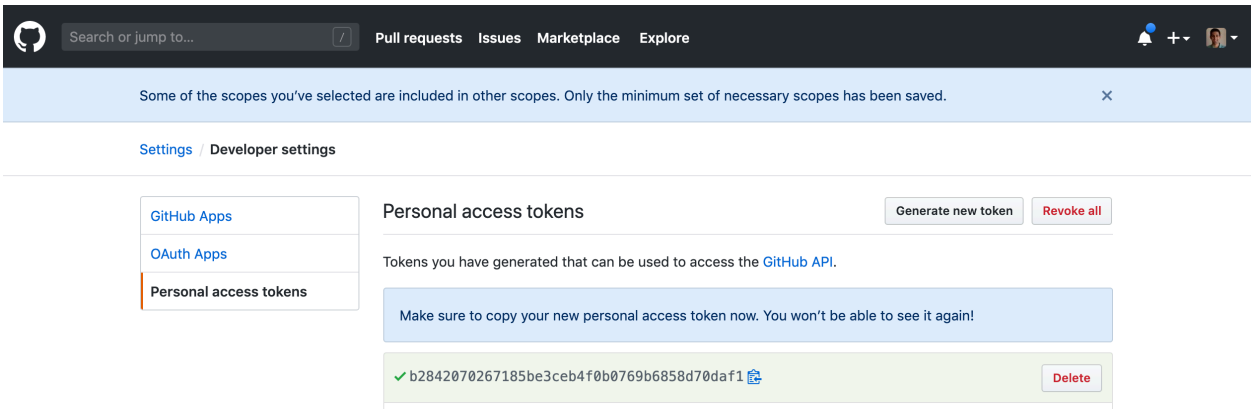

Ilustración 60 Token generado

Una vez obtenido el token a través de la interfaz de GitHub, se puede volver a GitLab para continuar con la importación del proyecto, para ello se selecciona la opción "Import project" y a continuación "GitHub" como fuente de estos repositorios.

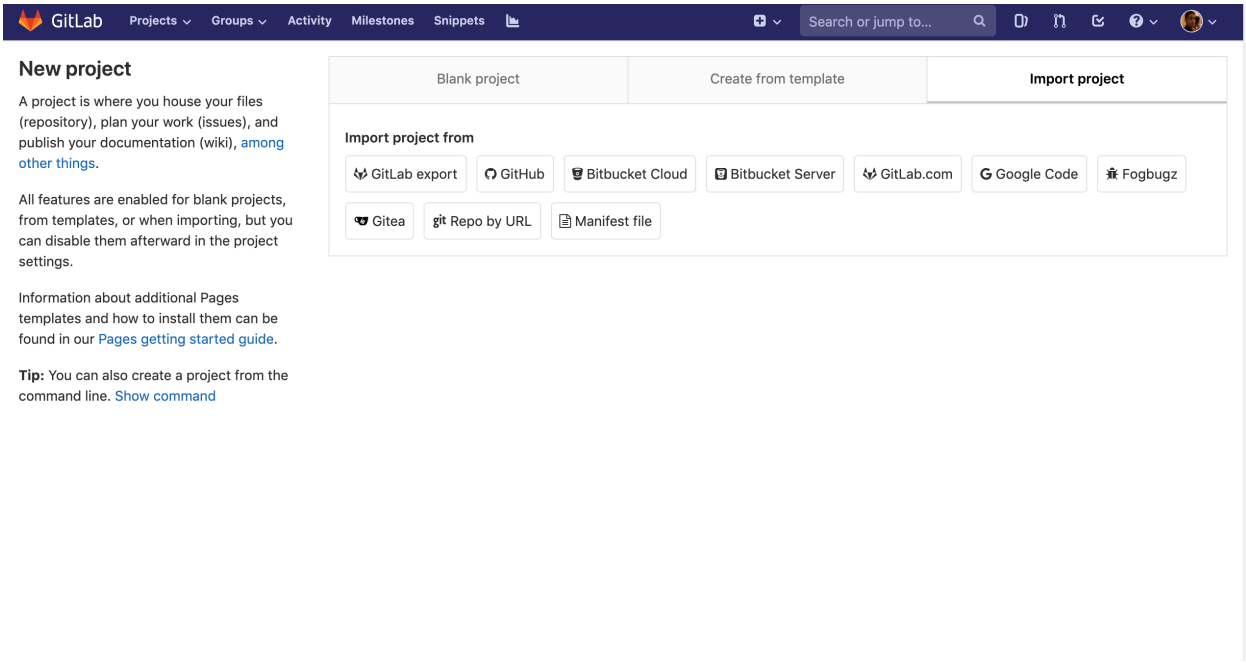

Ilustración 61 Crear nuevo proyecto importado de GitHub

Tras pegar el token obtenido previamente y hacer click en “List your GitHub repositories”, aparece una lista con todos los repositorios de GitHub. Tras cargar la lista (lo cual puede tardar cierto tiempo según la cantidad de repositorios existentes), aparecen los diferentes repositorios que se pueden importar desde GitHub, es posible importar tantos como se quieran. Al comienzo de la lista hay un botón para importarlos todos sin tener que seleccionar uno a uno.

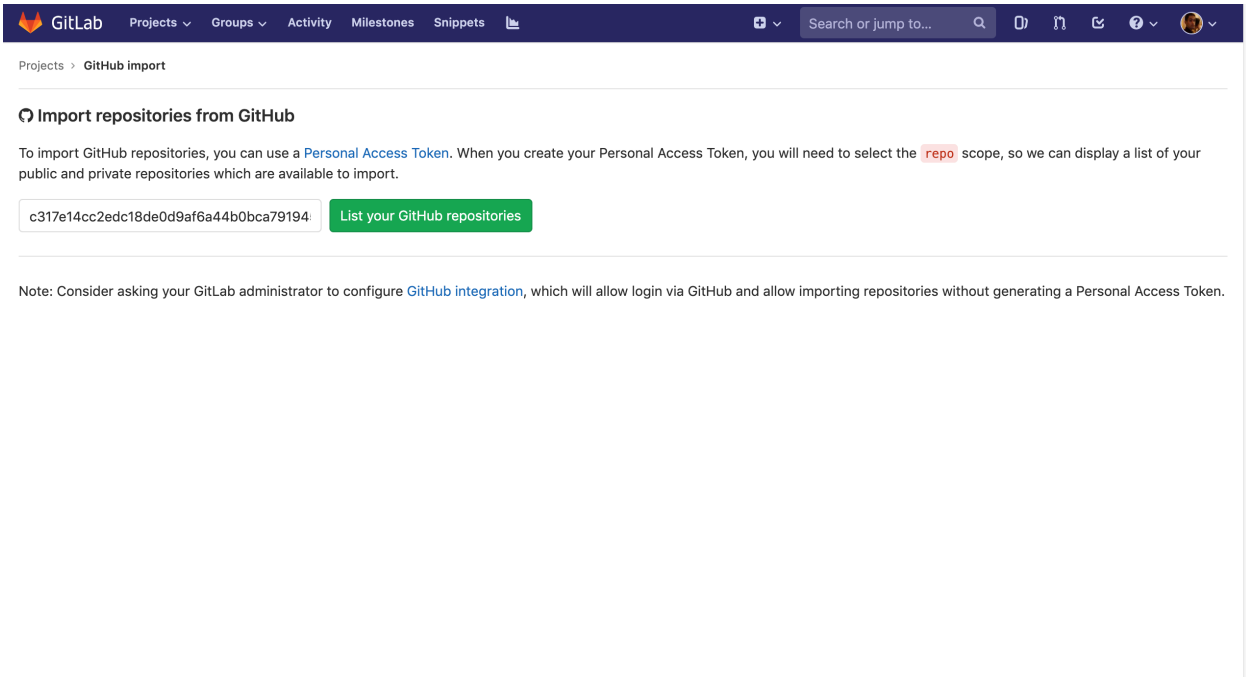

Ilustración 62 Token creado en GitHub y pegado en GitLab

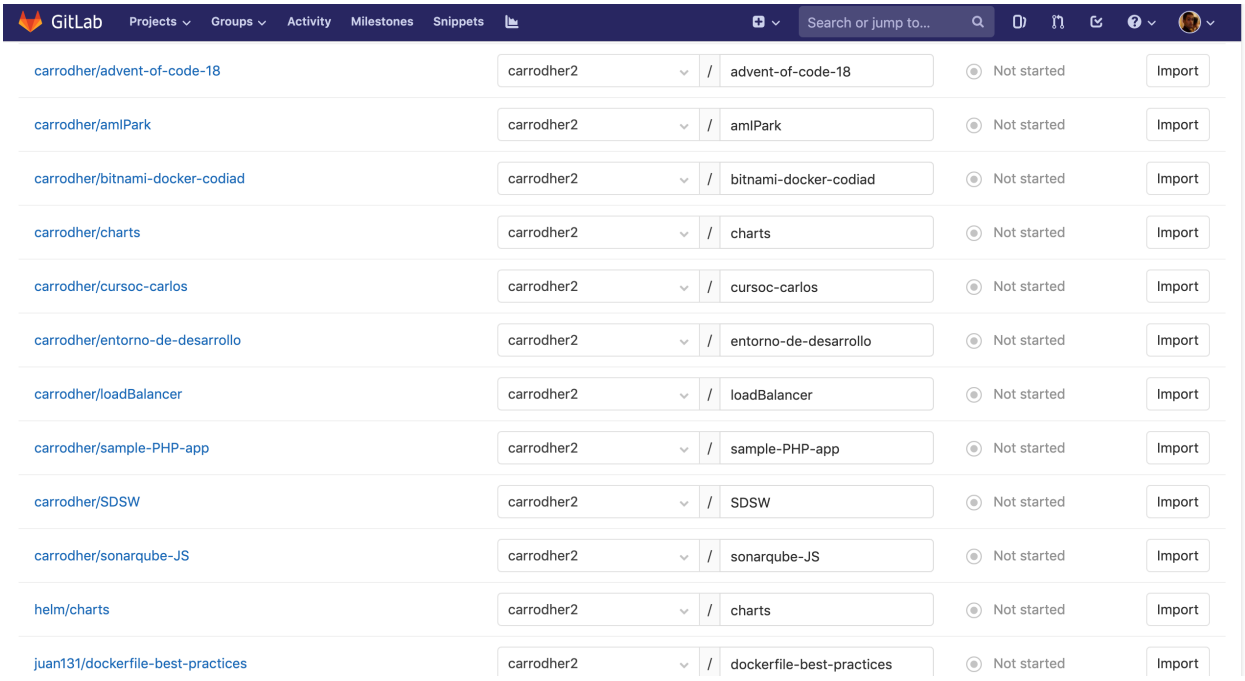

Ilustración 63 Lista de proyectos importables

En este caso, siguiendo los ejemplos del apartado anterior, se importará el proyecto *sonarqube-JS*.

Una vez importado es posible acceder al entorno de desarrollo, eliminar el proyecto existente que fue clonado desde GitHub y clonarlo de nuevo usando la URL del servidor GitLab.

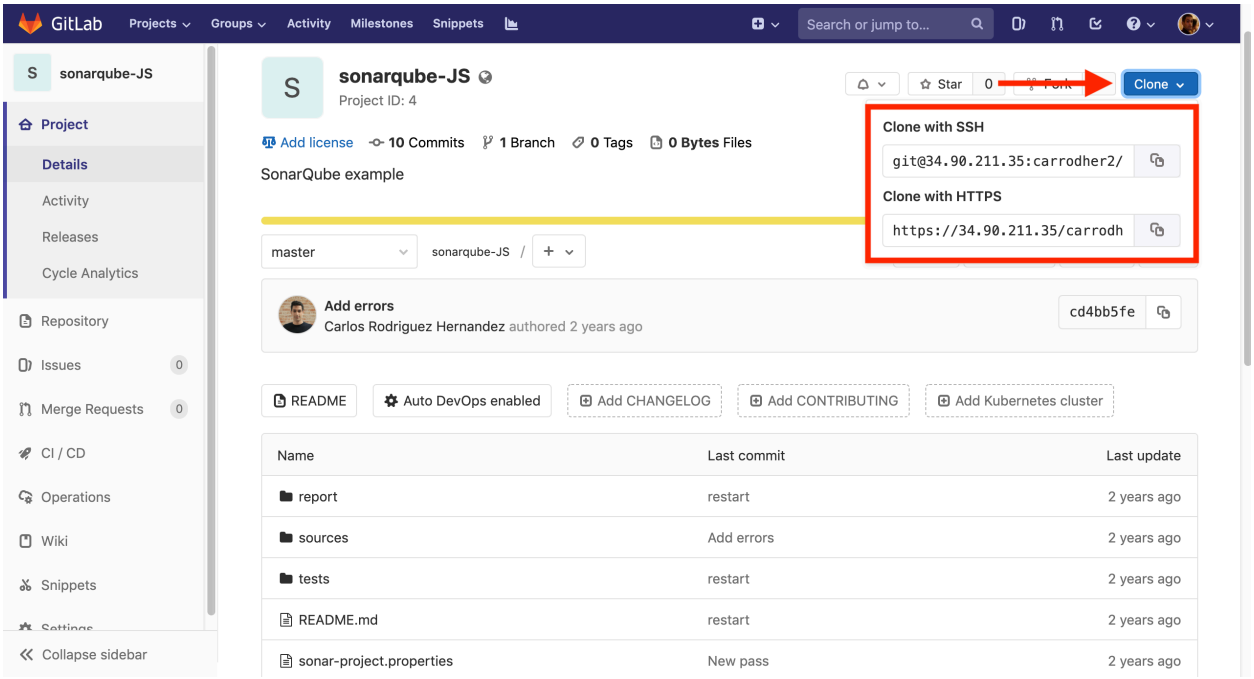

Ilustración 64 Proyecto importado y dirección para clonarlo

Una vez que se tienen los proyectos clonados en el entorno de desarrollo y configurados para que Git apunte a la URL del servidor GitLab (se hace automáticamente al clonar usando dicha dirección HTTPS), es posible seguir un flujo de trabajo habitual basado en Git para desarrollar en estos proyectos.

De esta forma es posible crear ramas, subir las ramas a GitLab y como se verá a continuación solicitar revisión del código por parte de otros compañeros del equipo antes de incluir estos cambios en la rama máster o cualquier otra rama que se entienda como producción o estable, esto se hace mediante *Merge requests*.

Tareas

Como se explicó en la introducción teórica sobre GitLab, también es posible gestionar tareas (issues) asociadas a un proyecto, de esta manera los desarrolladores tienen control sobre el estado del proyecto, qué tareas están en progreso, acabadas, planeadas para el futuro, etc.

Estas tareas pueden ser creadas por cualquier usuario con permisos para el proyecto, habrá ocasiones en las que sea el propio desarrollador quien crea una tarea, otras veces vendrá de otros compañeros de equipo o incluso de otros roles como puede ser el mánager o equipos de producto o negocio.

En este ejemplo se va a crear una tarea que requiere modificar el código del proyecto *sample-PHP-app* para así enlazar con la siguiente sección y mostrar el flujo de trabajo extremo a extremo, desde la creación y clonado del repositorio visto en la sección anterior hasta la creación de una revisión de código para implementar cambios pasando por la creación de una tarea solicitando estos cambios en el repositorio.

Para crear una tarea se puede hacer mediante la barra lateral que hay en la vista principal del proyecto. En este caso se va a solicitar la modificación de un fichero *html* existente en el repositorio

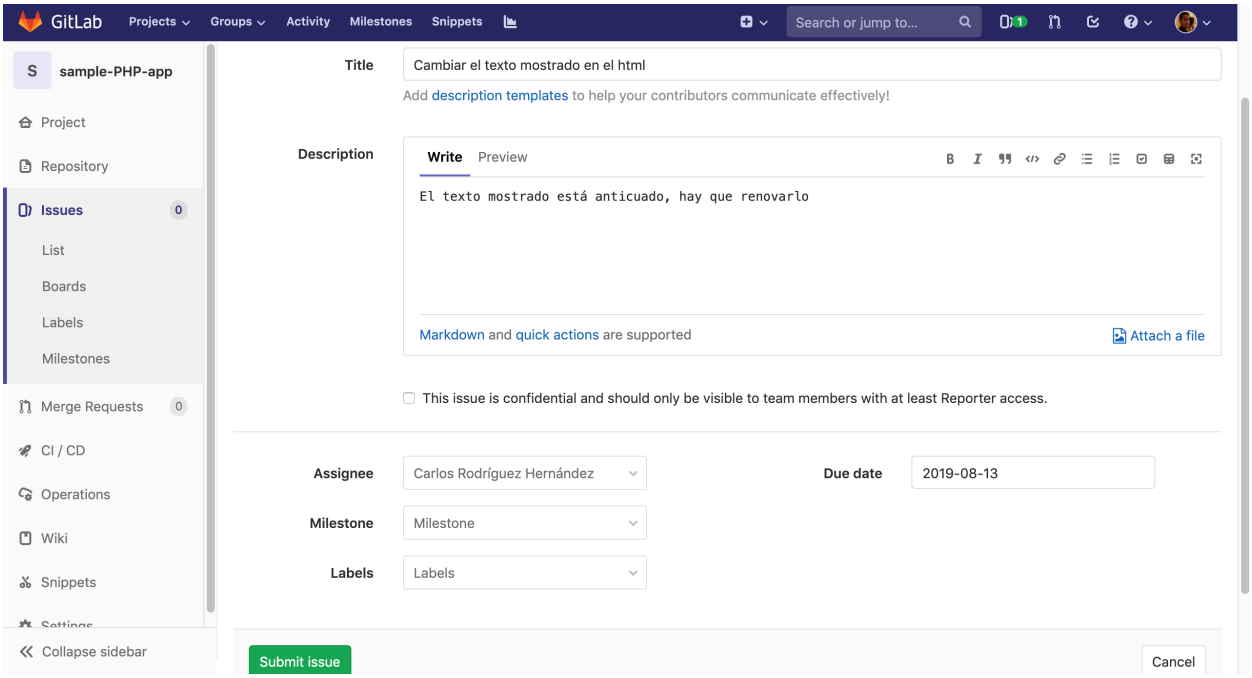

Ilustración 65 Crear nuevo issue con información sobre el cambio, asignación, fecha, etc.

Una vez creado el issue se puede acceder al tablero donde se organizan las diferentes tareas y moverlas entre las diferentes columnas. En este punto es posible crear diferentes columnas y tableros según las necesidades del equipo de trabajo.

Un tablero es la representación gráfica del estado de un proyecto donde se encuentran todas las tareas independientemente del estado en el que se encuentren dichas tareas. Cada estado es representado por las columnas. Depende de la metodología seguida por el equipo de trabajo, estos estados pueden variar; para este ejemplo se ha usado un tablero con tres columnas: *To Do*, *Doing* y *Closed*. Estos estados se corresponden con la situación en la que dicha tarea se encuentra. Un tablero es algo dinámico mientras se trabaja en un proyecto, puesto que las tareas cambian de un estado a otro según el trabajo del equipo, así como los objetivos marcados.

Para este ejemplo se mueve la tarea en cuestión a la columna *Doing* puesto que se va a empezar a trabajar en

ella, de igual manera accediendo al issue se puede ver información sobre la tarea, debatir con otros miembros del equipo mediante comentarios, establecer fecha para realizarla, asignarla a un desarrollador, etc.

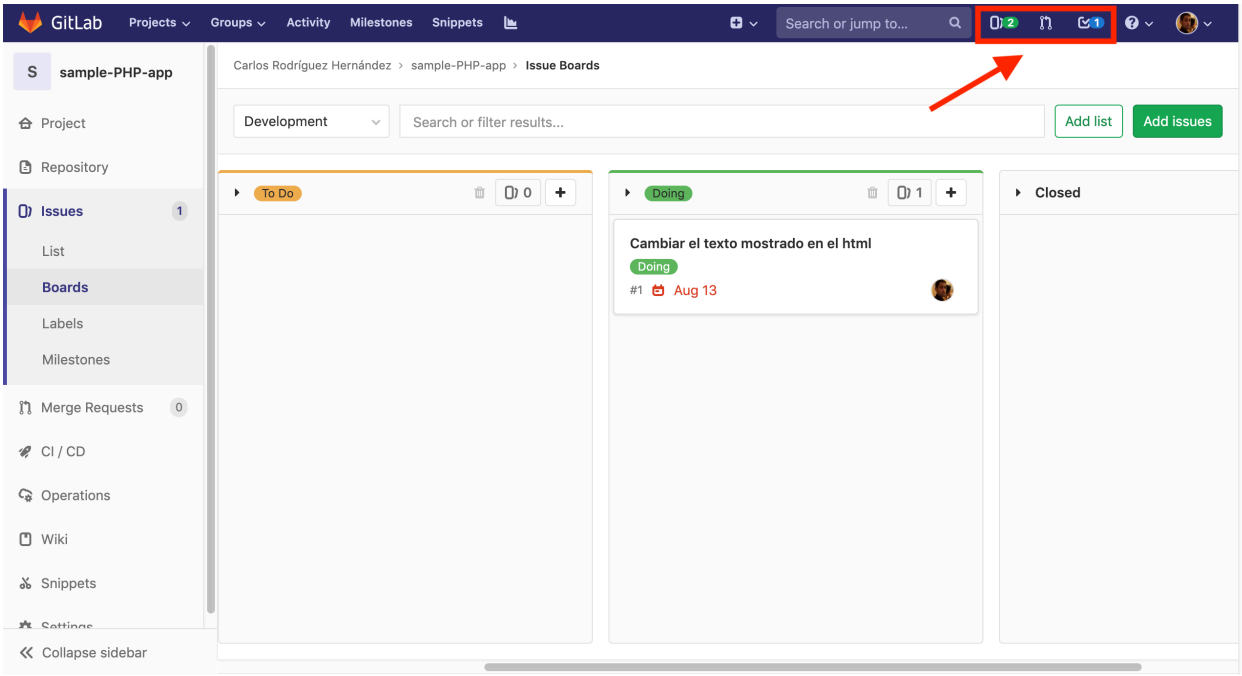

Ilustración 66 Issue en *Doing* y notificaciones

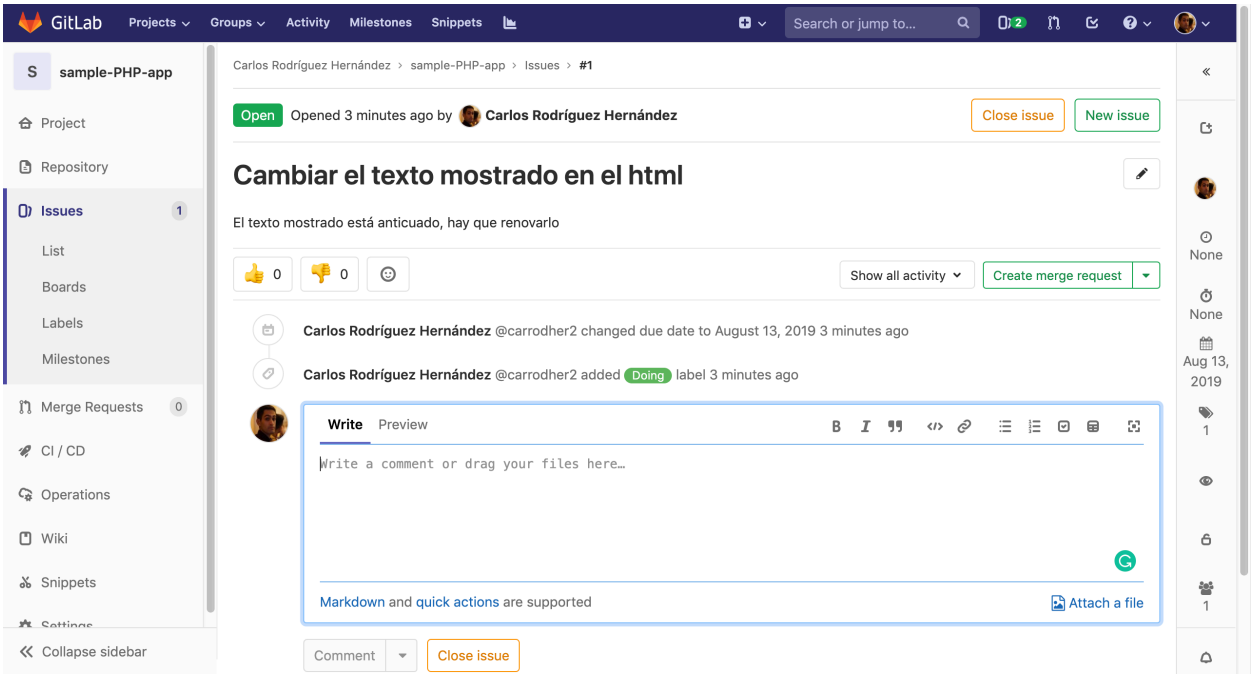

Ilustración 67 Detalles de la tarea

Merge Requests

Los merge request es la forma de enviar revisiones de código antes de incluir cambios en una rama que se considere máster, producción o fuente del código estable. Esto no es necesario en ningún caso, el desarrollador puede incluir su código en cualquier rama simplemente haciendo `git push origin <branch>` a cualquiera de las ramas disponibles en Git, siempre que dicho desarrollador tenga los permisos adecuados para el repositorio

en cuestión.

Aunque no es obligatorio, es buena práctica crear una rama nueva cuando se quiere trabajar en una nueva funcionalidad, realizar los cambios deseados en dicha rama y subir esta rama a GitLab para solicitar la revisión del código por parte de otros miembros del equipo antes de incluir estos cambios en la rama estable.

En caso de que el proyecto disponga de tests y se haya configurado el sistema de CI/CD, es frecuente ejecutar dichos tests automáticos cuando se crea un merge request, puesto que así se está comprobando que los nuevos cambios pasan los tests establecidos. Aparte de esta política, también es posible añadir ejecutar tests de manera periódica, cada vez que haya un nuevo commit, etc. Como ya se ha comentado, para ello el repositorio en cuestión debe disponer de los tests necesarios, aunque GitLab proporciona las herramientas y la infraestructura para hacerlo, es un trabajo que debe partir del equipo de desarrolladores o de la compañía en cuestión.

Siguiendo el ejemplo anterior, se va a mostrar cómo sería el flujo de trabajo una vez que un desarrollador recibe una tarea que necesita modificar el código. Se parte de una situación en la que el desarrollador ha accedido al entorno de desarrollo y dispone del repositorio clonado en su carpeta *projects* y configurado con GitLab como se ha visto en la sección anterior.

Los cambios se pueden realizar usando cualquier editor de texto o IDE desde la máquina local del desarrollador, aunque si el cambio no requiere mucho esfuerzo es más rápido hacerlo desde la propia línea de comandos del entorno de desarrollo usando un editor de texto como *Vim*.

```
## Crear nueva rama
root:~/projects/sample-PHP-app
▶ git checkout -b tareaTexto

## Implementar los cambios
root:~
▶ vim index.html

## Revisar los cambios
root:~/projects/sample-PHP-app
▶ git diff
diff --git a/index.html b/index.html
index 14f8658..3e25853 100644
--- a/index.html
+++ b/index.html
@@ -1 +1 @@
-<h1> This is an HTML test </h1>
+<h1> This is the new HTML text required by my manager </h1>

## Añadir los cambios, hacer el commit y subirlos a la rama
root:~/projects/sample-PHP-app
▶ git add index.html

root:~/projects/sample-PHP-app
▶ git commit -m "Cambiar texto"

root:~/projects/sample-PHP-app
▶ git push origin tareaTexto
```

Bloque de código 38 Flujo de trabajo para hacer cambios en el proyecto

En la vista principal del proyecto que se muestra en la siguiente ilustración, se puede ver que aparecen dos ramas, haciendo click en la nueva rama creada se pueden revisar los ficheros del último asentamiento de esta. Desde esta página es posible crear el merge request pulsando el botón “Create merge request”.

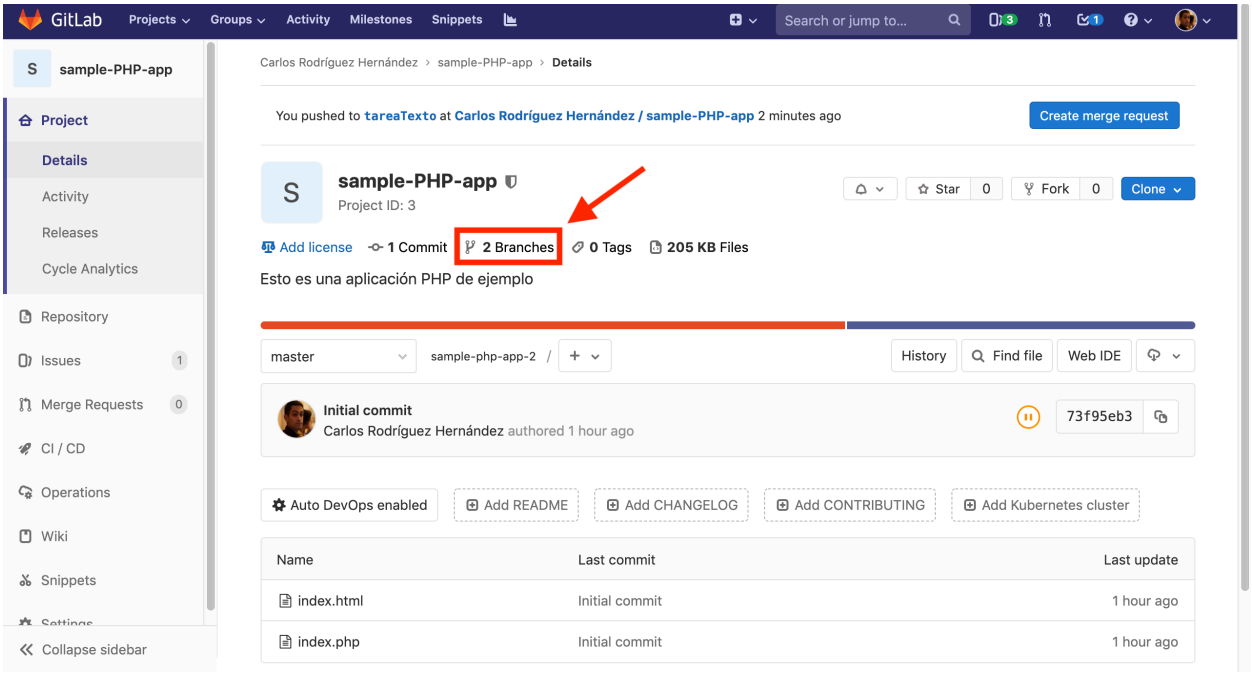

Ilustración 68 Vista del proyecto con dos ramas

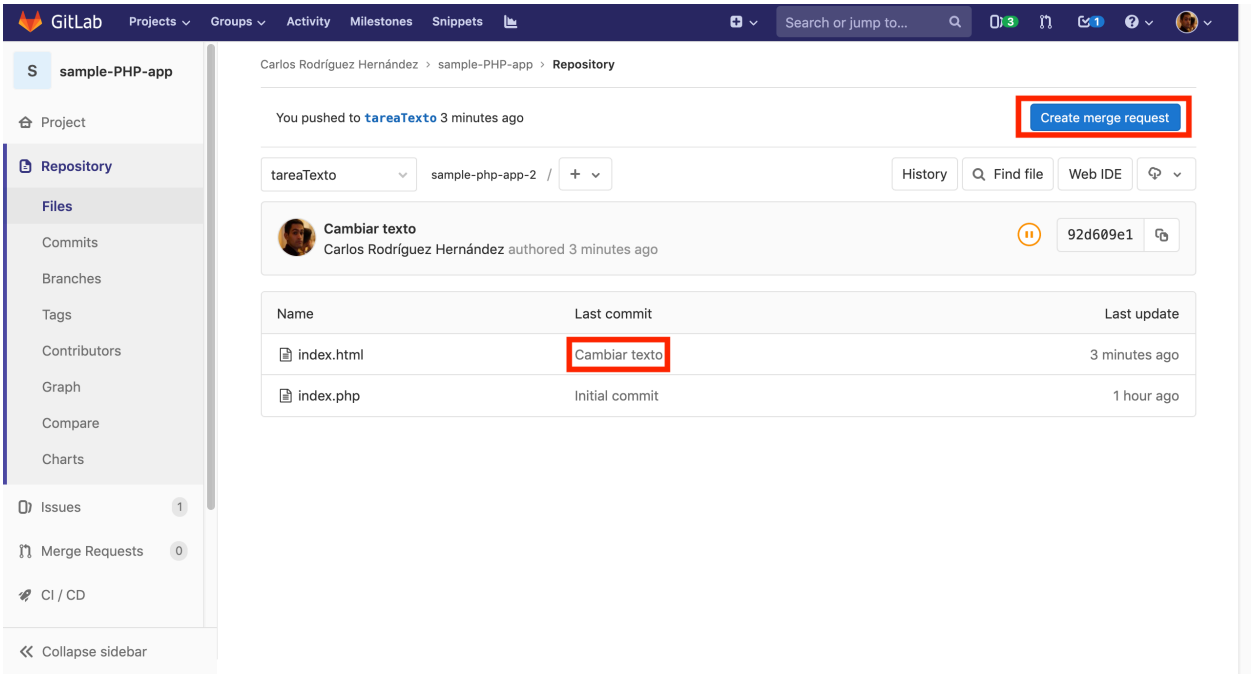

Ilustración 69 Estado del proyecto en la nueva rama

Al hacer click en “Create merge request” aparece una nueva vista en la que es posible seleccionar algunos parámetros como la rama desde la que queremos hacer el merge, asignarlo a una persona para que revise el código, etc.

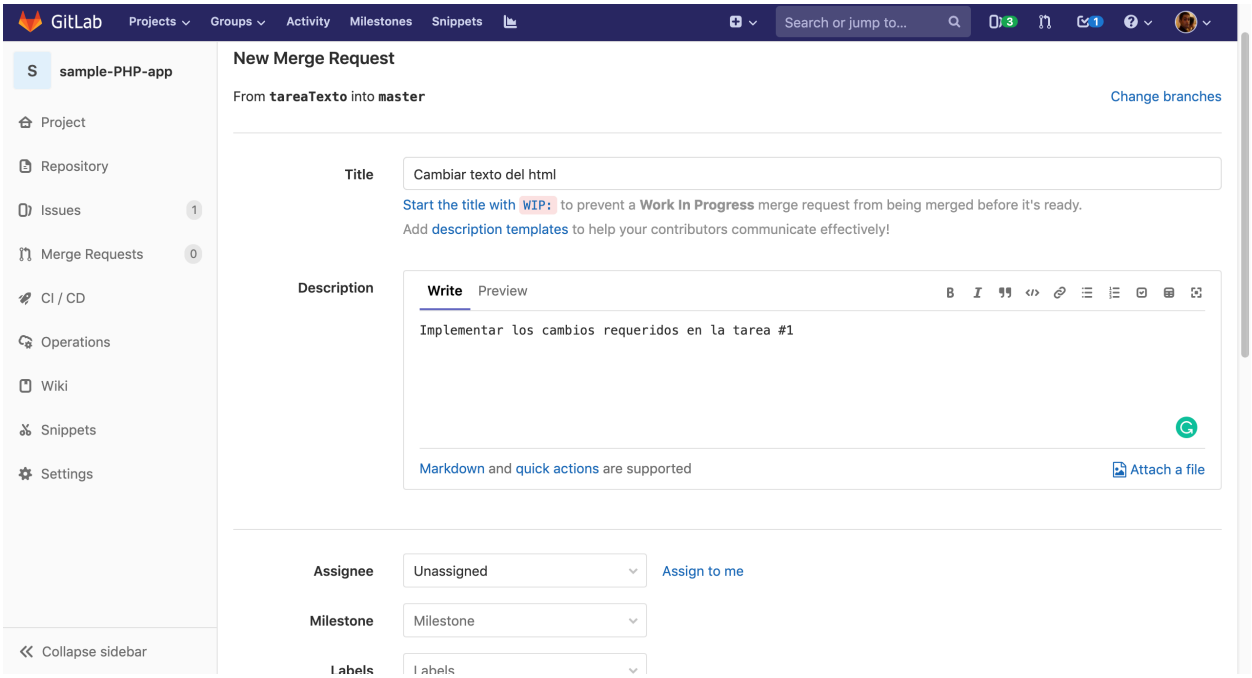

Ilustración 70 Crear merge request

Una vez creado el merge request, el desarrollador o miembro del equipo al que se le ha asignado la tarea de revisión puede realizar dicha tarea. Para lo cual puede hacer uso de comentarios, sugerencias y comentarios en línea en los propios ficheros modificados.

El desarrollador que ha creado el merge request puede aplicar todos los cambios necesarios siguiendo el proceso anterior, cada vez que se haga `git push` a la rama con los cambios el merge request es actualizado automáticamente y los revisores ven los nuevos cambios.

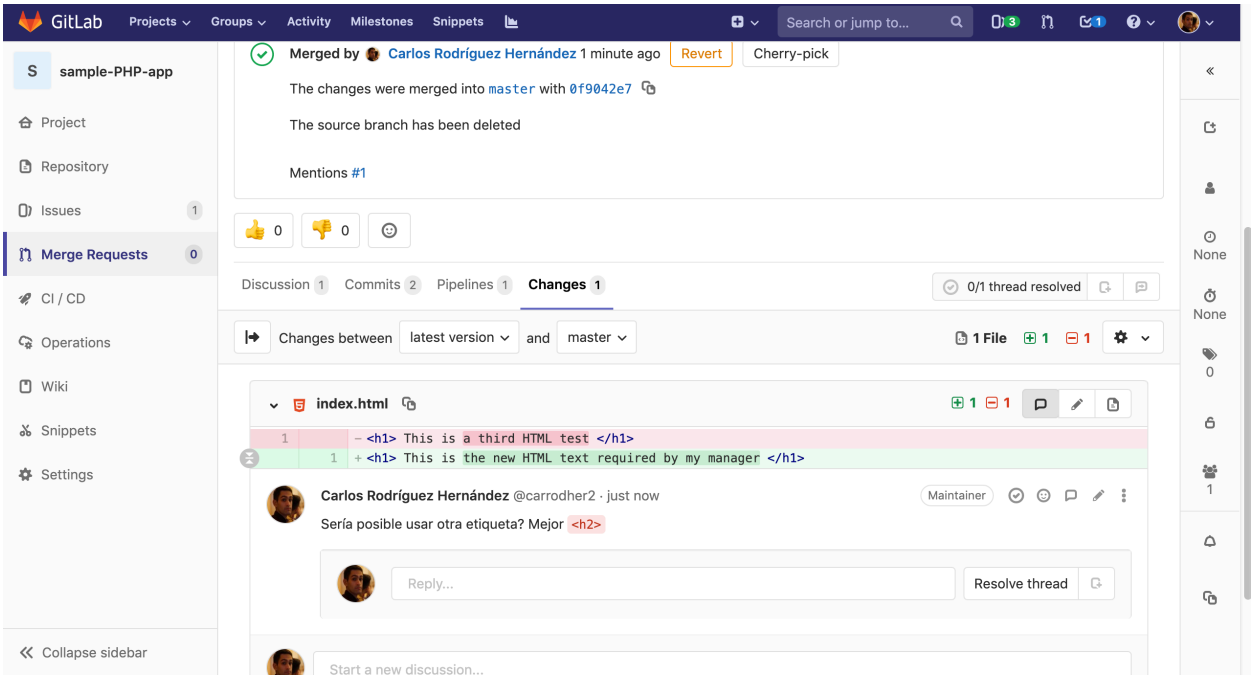

Ilustración 71 Revisión de código

Cuando el proceso de revisión haya finalizado, el merge request se puede finalizar y los cambios son fusionados con la rama elegida, estando así disponibles en dicha rama. Como se puede ver en la siguiente imagen, ya únicamente aparece una rama (la rama *tareaTexto* ha sido eliminada al finalizar el merge request) y los cambios han sido añadidos a la rama máster.

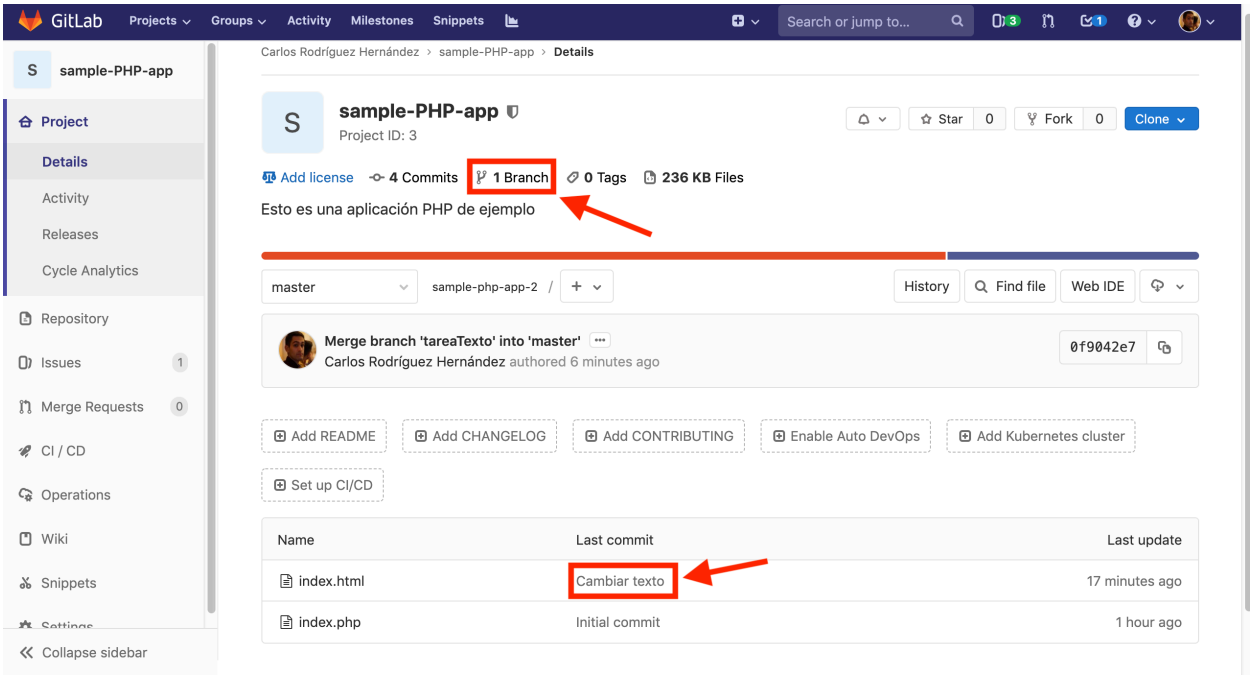

Ilustración 72 Estado del proyecto tras finalizar el merge request

Accediendo al propio fichero desde la interfaz web se puede apreciar el cambio realizado, de igual manera es posible editar el fichero de manera online e incluso GitLab dispone de un IDE online para realizar modificaciones en el repositorio sin necesidad de usar la línea de comandos para los comando de Git o el editor/IDE local. Esto es útil para realizar pequeñas modificaciones, no desarrollos completos.

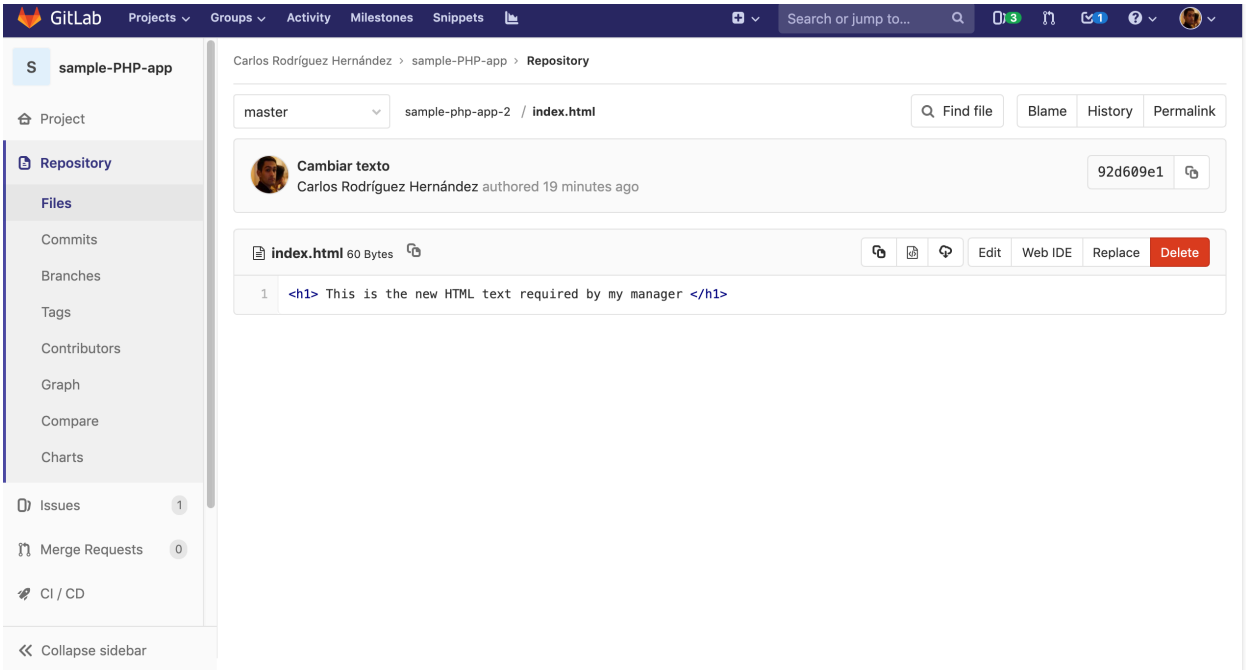

Ilustración 73 Vista del fichero con el cambio y opciones para editarlo

5 CONCLUSIONES

A lo largo de este proyecto se ha diseñado, desplegado y configurado de manera experimental un entorno de desarrollo para un equipo de trabajo distribuido. Este entorno de desarrollo es fácilmente escalable y, por tanto, válido para equipos de diferentes tamaños. También ha sido diseñado teniendo en cuenta la pluralidad de roles que pueden formar un mismo equipo.

Para ello se ha diseñado un sistema basado en contenedores Docker, proporcionando una imagen altamente personalizable con diversas herramientas para desarrollar a nivel local; de la misma forma se ha instalado y configurado una herramienta centralizada denominada GitLab que sirve como alojamiento para los diferentes repositorios de código así como para la gestión de las tareas asociadas a un proyecto software, por tanto, esta herramienta no solo está orientada a los desarrolladores sino también a otros roles dentro de un equipo de trabajo.

De este proyecto se sacan diversas enseñanzas a nivel técnico como pueden ser la construcción de imágenes basadas en Docker, la configuración de Git o instalación de GitLab. Se tratan en el proyecto los diferentes componentes implicados en el desarrollo de un producto software, la variedad de herramientas que entran en acción, así como la importancia de una correcta política de buenas prácticas a la hora de desarrollar un producto software.

El entorno diseñado en este proyecto pretende facilitar el cumpliemento de una serie de buenas prácticas a la hora de la gestión del ciclo de vida del software que ayudan a garantizar la calidad del software.

Mediante este proyecto, se pretende facilitar la implementación de estas políticas, haciéndolo accesible a cualquier equipo de trabajo mediante una mínima configuración.

Quizás la principal dificultad encontrada en este proyecto ha sido conseguir un entorno útil para el mayor número de equipos posible con una imagen que no requiera un elevado esfuerzo de personalización para adapatarse a un equipo en particular. Esto ha sido resuelto mediante una configuración básica común a la mayoría de los equipos de desarrollo (Git, poder usar el propio IDE, runtimes de los lenguajes más comunes instalados, etc.) y además otorgando la posibilidad de personalizar la imagen de manera fácil mediante scripts ejecutados en diferentes momentos del despliegue de dicha imagen.

La configuración diseñada pretende servir de ejemplo para la implementación de este entorno en cualquier equipo, necesitando, obviamente ciertas personalizaciones tanto a nivel del desarrollador local como del equipo completo.

REFERENCIAS

[1] D. Firesmith, «Virtualization via Containers,» [En línea]. Available: https://insights.sei.cmu.edu/sei_blog/2017/09/virtualization-via-containers.html.

[2] E. A., «Docker, Qué es y sus principales características.,» [En línea]. Available: https://openwebinars.net/blog/docker-que-es-sus-principales-caracteristicas/.

[3] GitLab Docs, «GitLab CI/CD,» [En línea]. Available: https://docs.gitlab.com/ce/ci/README.html.

[4] Rajkumar, «Software Development Life Cycle – SDLC | Software Testing Material,» [En línea]. Available: https://www.softwaretestingmaterial.com/sdlc-software-development-life-cycle/.

[5] techopedia, «Software Development Life Cycle (SDLC),» [En línea]. Available: https://www.techopedia.com/definition/22193/software-development-life-cycle-sdlc.

[6] CommentCaMarche, «Ciclo de vida del 'software',» [En línea]. Available: https://es.ccm.net/contents/223-ciclo-de-vida-del-software.

[7] J. H. Canós, P. Letelier y M. C. Penadés, «Métodologías Ágiles en el Desarrollo de Software,» [En línea]. Available: http://roa.ult.edu.cu/jspui/bitstream/123456789/476/1/TodoAgil.pdf.

[8] Wikipedia, «Comparison of issue-tracking systems,» [En línea]. Available: https://en.wikipedia.org/wiki/Comparison_of_issue-tracking_systems.

[9] bugzilla.org contributors, «Bugzilla main page,» [En línea]. Available: https://www.bugzilla.org/.

[10] MantisBT Team, «MantisBT main page,» [En línea]. Available: https://www.mantisbt.org/.

[11] GitLab, «All GitLab Features,» [En línea]. Available: https://about.gitlab.com/features/.

[12] Phacility, Inc., «Phabricator main page,» [En línea]. Available: https://www.phacility.com/.

[13] J.-P. Lang, «Redmine main page,» [En línea]. Available: https://www.redmine.org/.

[14] Atlassian, «What is version control,» [En línea]. Available: https://www.atlassian.com/git/tutorials/what-is-version-control.

[15] Tower, «What is Version Control?,» [En línea]. Available: https://www.git-tower.com/learn/git/ebook/en/command-line/basics/what-is-version-control.

[16] Tower, «Why Use a Version Control System?,» [En línea]. Available: https://www.git-tower.com/learn/git/ebook/en/command-line/basics/why-use-version-control.

[17] Tower, «Version Control Best Practices,» [En línea]. Available: https://www.git-tower.com/learn/git/ebook/en/command-line/appendix/best-practices.

[18] Git, «Getting Started - About Version Control,» [En línea]. Available: https://git-

scm.com/book/en/v2/Getting-Started-About-Version-Control.

[19] Git, «Distributed Git - Distributed Workflows,» [En línea]. Available: https://git-scm.com/book/en/v2/Distributed-Git-Distributed-Workflows.

[20] N. Kadivar, «Top 10 Version Control Systems,» [En línea]. Available: https://hackernoon.com/top-10-version-control-systems-4d314cf7adea.

[21] Git, «Getting Started - What is Git?,» [En línea]. Available: https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F.

[22] Ubuntu, «BasicChroot,» [En línea]. Available: https://help.ubuntu.com/community/BasicChroot.

[23] Aqua Blog, «Una Breve Historia de los Contenedores: desde los años 70 a Docker 2016,» [En línea]. Available: https://hoplasoftware.com/una-breve-historia-de-los-contenedores-desde-los-anos-70-a-docker-2016/.

[24] M. Riondato, «Chapter 14. Jails,» [En línea]. Available: https://www.freebsd.org/doc/handbook/jails.html.

[25] Linux VServer, «Overview Linux VServer,» [En línea]. Available: http://linux-vserver.org/Overview.

[26] P. Menage, «cgroups,» [En línea]. Available: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[27] M. Helsley, «LXC: Linux container tools,» [En línea]. Available: https://developer.ibm.com/tutorials/l-lxc-containers/.

[28] M. Zhylinski, «Cloud Foundry Containers: Warden, Docker, and Garden,» [En línea]. Available: https://www.altoros.com/blog/cloud-foundry-containers-warden-docker-and-garden/.

[29] lmctfy, «lmctfy - Let Me Contain That For You,» [En línea]. Available: https://github.com/google/lmctfy#lmctfy---let-me-contain-that-for-you.

[30] E. Carter, «2018 Docker usage report,» [En línea]. Available: https://sysdig.com/blog/2018-docker-usage-report/.

[31] Kubernetes, «What is Kubernetes,» [En línea]. Available: http://kubernetes.io/docs/whatisk8s/ .

[32] C. White, «Linux Isolation Basics,» [En línea]. Available: https://www.engineyard.com/blog/linux-containers-isolation.

[33] A. Sostre, «Containers vs Virtual Machines,» [En línea]. Available: https://www.serverpronto.com/spu/2016/05/containers-vs-virtual-machines-vms-is-there-a-clear-winner/.

[34] Isaac, «Docker: todo sobre los contenedores,» [En línea]. Available: https://www.linuxadictos.com/docker-i-que-es-conociendo-la-ballena.html.

[35] Docker Inc., «About Docker Engine,» [En línea]. Available: https://docs.docker.com/engine/.

[36] Docker, Inc., «Docker overview,» [En línea]. Available: https://docs.docker.com/engine/docker-overview/.

[37] rkt, «rkt - the pod-native container engine,» [En línea]. Available: https://github.com/rkt/rkt.

[38] UpGuard, «Docker vs CoreOS Rkt,» [En línea]. Available: https://www.upguard.com/articles/docker-vs-coreos.

[39] Apache, «Containerizers,» [En línea]. Available: http://mesos.apache.org/documentation/latest/containerizers/.

[40] Apache, «Containerizer internals,» [En línea]. Available: http://mesos.apache.org/documentation/latest/containerizer-internals/.

[41] okteto, «Get started in 5 minutes,» [En línea]. Available: https://okteto.com/docs/getting-started/index.html.

[42] GitLab, «A brief history of GitLab,» [En línea]. Available: https://about.gitlab.com/company/history/.

[43] GitLab, «Community Edition or Enterprise Edition,» [En línea]. Available: https://about.gitlab.com/install/ce-or-ee/?distro=ubuntu.

[44] GitLab, «GitLab Installation,» [En línea]. Available: https://about.gitlab.com/install/#debian.

[45] G. Docs, «Requirements,» [En línea]. Available: https://docs.gitlab.com/ee/install/requirements.html.

[46] GitLab Docs, «Configuration options,» [En línea]. Available: https://docs.gitlab.com/omnibus/settings/configuration.html.

[47] GitLab Docs, «Projects,» [En línea]. Available: https://docs.gitlab.com/ce/user/project/index.html.

[48] Í. Serrano, «GitLab #7: Gestión de tareas,» [En línea]. Available: http://www.inigoserrano.com/gitlab-7-gestion-de-tareas/.

[49] R. Torralba, «Gestión de tareas multiproyecto con Gitlab,» [En línea]. Available: https://www.artansoft.com/2017/11/gestion-de-tareas-gitlab/.

[50] Git, «Getting Started - A Short History of Git,» [En línea]. Available: https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git.

[51] R. Dudler, «git - the simple guide,» [En línea]. Available: https://rogerdudler.github.io/git-guide/.

[52] Git, «Getting Started - Installing Git,» [En línea]. Available: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git.

[53] Git, «Getting Started - First-Time Git Setup,» [En línea]. Available: https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup.

[54] Git, «Git Documentation,» [En línea]. Available: https://git-scm.com/doc.

[55] Tower, «Working on Your Project,» [En línea]. Available: https://www.git-

tower.com/learn/git/ebook/en/command-line/basics/working-on-your-project.

[56] Git, «Git Basics - Getting a Git Repository,» [En línea]. Available: https://git-scm.com/book/en/v2/Git-Basics-Getting-a-Git-Repository.

[57] Git, «Git Basics - Recording Changes to the Repository,» [En línea]. Available: https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository.

[58] Git, «Git Basics - Viewing the Commit History,» [En línea]. Available: https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History.

[59] Git, «Git Basics - Working with Remotes,» [En línea]. Available: https://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes.

[60] Tower, «Branching can Change Your Life,» [En línea]. Available: https://www.git-tower.com/learn/git/ebook/en/command-line/branching-merging/branching-can-change-your-life.

[61] Git, «Git Branching - Branches in a Nutshell,» [En línea]. Available: https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell.

[62] Git, «Git Branching - Basic Branching and Merging,» [En línea]. Available: https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging.

[63] Wikipedia, «Docker (software) - History,» [En línea]. Available: https://en.wikipedia.org/wiki/Docker_(software)#History.

[64] Wikipedia, «Docker, Inc. - History,» [En línea]. Available: https://en.wikipedia.org/wiki/Docker,_Inc..

[65] P. Belagatti, «This Is How BIG BOYS Are Using Kubernetes,» [En línea]. Available: https://www.linkedin.com/pulse/how-big-boys-using-kubernetes-pavan-belagatti/.

[66] S. P. Kane y K. Matthias, Docker: Up & Running: Shipping Reliable Containers in Production, O'Reilly, 2018.

[67] Docker, Inc., «Get Started, Part 1: Orientation and setup,» [En línea]. Available: https://docs.docker.com/get-started/.

[68] Docker, Inc., «Get Docker Engine - Community for Debian,» [En línea]. Available: https://docs.docker.com/install/linux/docker-ce/debian/.

[69] Docker Inc., «Get Docker Engine - Community for CentOS,» [En línea]. Available: https://docs.docker.com/install/linux/docker-ce/centos/.

[70] Docker Inc., «Get Docker Engine - Community for Fedora,» [En línea]. Available: https://docs.docker.com/install/linux/docker-ce/fedora/.

[71] Docker, Inc., «Post-installation steps for Linux,» [En línea]. Available: https://docs.docker.com/install/linux/linux-postinstall/.

[72] Docker, Inc., «Get Started, Part 2: Containers,» [En línea]. Available: https://docs.docker.com/get-started/part2/.

[73] P. Srivastav, «Docker for beginners,» [En línea]. Available: https://docker-curriculum.com/.

[74] S. Goasguen, Docker Cookbook, O'Reilly, 2015.

[75] A. Mouat, Using Docker, O'Reilly, 2015.

[76] T. Lovett, «Getting Started with Docker,» [En línea]. Available: https://scotch.io/tutorials/getting-started-with-docker.

[77] Alpine Linux, «Alpine Linux,» [En línea]. Available: https://www.alpinelinux.org/about/.

[78] C. R. Hernández, «TFG-DevEnv,» [En línea]. Available: https://github.com/carrodher/entorno-de-desarrollo.

[79] SourceForge, «Procps - The /proc file system utilities,» [En línea]. Available: http://procps.sourceforge.net/.

[80] Vim community, «Vim - the ubiquitous text editor,» [En línea]. Available: https://www.vim.org/.

[81] A. Heddings, «What is ZSH, and Why Should You Use It Instead of Bash?,» [En línea]. Available: https://www.howtogeek.com/362409/what-is-zsh-and-why-should-you-use-it-instead-of-bash/.

[82] Free Software Foundation, Inc., «Introduction to GNU Wget,» [En línea]. Available: https://www.gnu.org/software/wget/.

[83] G. Greer, «The Silver Searcher,» [En línea]. Available: https://github.com/ggreer/the_silver_searcher.

[84] Microsoft, «Visual Studio Code,» [En línea]. Available: https://code.visualstudio.com/.

[85] Docker, Inc., «FROM instruction,» [En línea]. Available: https://docs.docker.com/engine/reference/builder/#from.

[86] Bitnami, «minideb base image,» [En línea]. Available: https://github.com/bitnami/minideb-extras.

[87] J. Westby, «minideb: a new container base image,» [En línea]. Available: https://engineering.bitnami.com/articles/minideb-a-new-container-base-image.html.

[88] Docker, Inc., «LABEL instruction,» [En línea]. Available: https://docs.docker.com/engine/reference/builder/#label.

[89] Docker, Inc., «RUN instruction,» [En línea]. Available: https://docs.docker.com/engine/reference/builder/#run.

[90] Docker, Inc., «COPY instruction,» [En línea]. Available: https://docs.docker.com/engine/reference/builder/#copy.

[91] Docker, Inc., «ENV instruction,» [En línea]. Available: https://docs.docker.com/engine/reference/builder/#env.

[92] Docker, Inc., «WORKDIR instruction,» [En línea]. Available: https://docs.docker.com/engine/reference/builder/#workdir.

[93] Docker, Inc., «ENTRYPOINT instruction,» [En línea]. Available: https://docs.docker.com/engine/reference/builder/#entrypoint.

[94] Docker, Inc., «CMD instruction,» [En línea]. Available: https://docs.docker.com/engine/reference/builder/#cmd.

[95] R. Russell, «Oh My Zsh,» [En línea]. Available: https://ohmyz.sh/.

[96] R. Russell, «oh-my-zsh themes,» [En línea]. Available: https://github.com/robbyrussell/oh-my-zsh/wiki/Themes.

[97] R. Russell, «oh-my-zsh plugins,» [En línea]. Available: https://github.com/robbyrussell/oh-my-zsh/wiki/Plugins.

[98] E. Freese, «zsh-autosuggestions plugin,» [En línea]. Available: https://github.com/zsh-users/zsh-autosuggestions.

[99] D. Shahaf, «zsh-syntax-highlighting plugin,» [En línea]. Available: https://github.com/zsh-users/zsh-syntax-highlighting.

[100] Bitnami, «Bitnami GCP launchpad guide,» [En línea]. Available: https://docs.bitnami.com/google/get-started-launchpad.

[101] Bitnami, «Step 1: Register With Google Cloud Platform,» [En línea]. Available: https://docs.bitnami.com/google/get-started-launchpad/#step-1-register-with-google-cloud-platform.

[102] Bitnami, «Step 2: Enable The Google Compute Engine API,» [En línea]. Available: https://docs.bitnami.com/google/get-started-launchpad/#step-2-enable-the-google-compute-engine-api.

[103] Bitnami, «Step 3: Register With Bitnami,» [En línea]. Available: https://docs.bitnami.com/google/get-started-launchpad/#step-3-register-with-bitnami.

[104] Bitnami, «Step 4: Connect Your Google Cloud Platform And Bitnami Accounts,» [En línea]. Available: https://docs.bitnami.com/google/get-started-launchpad/#step-4-connect-your-google-cloud-platform-and-bitnami-accounts.

[105] Bitnami, «Bitnami GCP launchpad,» [En línea]. Available: https://google.bitnami.com/.